Project 2 Report

Taylor Berger, Zachary Friedland, Jianyu Yang

December 13, 2014

1 Language Decisions

Our group continued to use Python for the ease of expressing high level concepts and removal of memory management from the project. We also used Git for version control the collaboration tools. The repository can be found at Taylor's Github.

We elected to use Zach's homework since he integrated the support of ANTLR and its powerful parsing constructs to make constructing the AST an easy task. We also took advantage of PyGraphViz for the AST and CFG visualization tasks.

2 Scanning, parsing and displaying the AST

Scanning, parsing and constructing the AST was done using ANTLR and the grammar found in the `simple.g` file displayed below.

```
grammar simple;

options{
        language=Python;
        output=AST;
        ASTLabelType=CommonTree;
        backtrack=true;
}
/* Arithmetic operators, in order of precedence. */
MULT        :       '*';
PLUS        :       '+';
MINUS       :       '-';

/* Boolean operators, in order of precedence. */
NOT         :       'not';
AND         :       '&';
OR          :       '|';
RELOP       :       ('=' | '<' | '<=' | '>' | '>=');
```

```
/* If / else. */
IF        :        'if';
THEN        :        'then';
ELSE        :        'else';
ENDIF        :        'fi';
SKIP        :        'skip';

/* Do while. */
WHILE        :        'while';
DO        :        'do';
ENDWHILE        :        'od';

/* Misc. */
GETS        :        ':=';
SEMI        :        ';';
LPAREN        :        '(';
RPAREN        :        ')';
BLOCK        :        'block';
UNARY        :        'unary';

/* Atoms. */
BOOLEAN        :        ('true' | 'false');
IDENT        :        ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9')*;
INTEGER        :        ('0'..'9')+;

/* Ignore whitespace. */
WS        :        (' ' | '\t' | '\n' | '\r' | '\f')+ {$channel = HIDDEN;};


program
        :        block
        ;

block
        :        statement+
                -> ^(BLOCK statement+)
        ;

/* Arithmetic expressions - craziness due to precendence! */
arith_expr
        :        add_expr
        ;
add_expr
        :        sub_expr (PLUS^ sub_expr)*
        ;
sub_expr
```

```
        :       mult_expr (MINUS^ mult_expr)*
        ;
mult_expr
        :       unary_expr (MULT^ unary_expr)*
        ;
unary_expr
        :       MINUS arith_atom
                -> ^(UNARY MINUS arith_atom)
        |       PLUS arith_atom
                -> ^(UNARY PLUS arith_atom)
        |       arith_atom
        ;
arith_atom
        :       (IDENT | INTEGER)
        |       LPAREN! arith_expr RPAREN!
        ;

/* Boolean expressions - craziness due to precedence! */
bool_expr
        :       or_expr
        ;
or_expr
         :       and_expr (AND^ and_expr)*
         ;
and_expr
         :       bool_atom (OR^ bool_atom)*
         ;
bool_atom
         :       BOOLEAN
        |       NOT^ bool_atom
        |       LPAREN! bool_expr RPAREN!
        |       arith_expr RELOP^ arith_expr
        ;

statement
    :   IDENT GETS^ arith_expr SEMI!
    |   SKIP SEMI!
    |   IF^ bool_expr THEN! block ELSE! block ENDIF! SEMI!
    |   WHILE^ bool_expr DO! block ENDWHILE! SEMI!
    ;
```

Since the code to parse and generate the AST is auto-generated code, it will be omitted for now, but can be found in the code appendix at the end of the report.

3 Decorated AST

4 Conversion to a CFG

Since displaying the graph was done through pygraphviz, we chose to directly convert the AST given to us by ANTLR directly into an AGraph object in pygraphviz. The following snippet expects an AST previously parsed by ANTLR and converts it into an AGraph that represents the CFG.

```
def convert_ast_to_cfg(ast):
    graph = AGraph()
    id = 0
    program = ""
    revers_stack = []
    for x in ast.children:
        revers_stack.append(x)
    stack = []
    while(len(revers_stack) != 0):
        stack.append(revers_stack.pop())
    layer_stack = []
    last_id = 0
    un_solved_stack = []
    while(len(stack) != 0):
        x = stack.pop()
        print x.text
        print un_solved_stack
        if x.text == ":=" or x.text == "skip":
            if x.text == "skip":
                program = "skip";
            else:
                program = str(x.children[0].text) + " := " + render_short(x.children[1])
            graph.add_node(id, l = program)
            if(last_id != 0):
                graph.add_edge(last_id, id)
            last_id = id
        elif x.text == "if":
            program = "if " + render_short(x.children[0])
            graph.add_node(id, l = program)
            if(last_id != 0):
                graph.add_edge(last_id, id)
            fi_node = copy(x.children[0])
            else_node = copy(x.children[2])
            then_node = copy(x.children[1])
            fi_node.token.text = "fi"
            else_node.token.text = "else"
            then_node.token.text = "then"
            stack.append(fi_node)
```

```
        stack.append(else_node)
        stack.append(then_node)
        layer_stack.append(("if", id))
        last_id = id
    elif x.text == "then":
        un_solved_stack.append("then")
        revers_stack = []
        for xx in x.children:
            revers_stack.append(xx)
        while(len(revers_stack) != 0):
            stack.append(revers_stack.pop())
    elif x.text == "else":
        un_solved_stack.pop()
        un_solved_stack.append(last_id)
        un_solved_stack.append("else")
        revers_stack = []
        for xx in x.children:
            revers_stack.append(xx)
        while(len(revers_stack) != 0):
            stack.append(revers_stack.pop())
        (statement, last_id) = layer_stack.pop()
        layer_stack.append((statement,last_id))
    elif x.text == "fi":
        un_solved_stack.pop()
        un_solved_stack.append(last_id)
        layer_stack.pop()
        last_id = 0
    elif x.text == "while":
        program = "while " + render_short(x.children[0])
        graph.add_node(id, l = program)
        if(last_id != 0):
            graph.add_edge(last_id,id)
        od_node = copy(x.children[0])
        do_node = copy(x.children[1])
        od_node.token.text = "od"
        do_node.token.text = "do"
        stack.append(od_node)
        stack.append(do_node)
        layer_stack.append(("while", id))
        last_id = id
    elif x.text == "do":
        revers_stack = []
        for xx in x.children:
            revers_stack.append(xx)
        while(len(revers_stack) != 0):
            stack.append(revers_stack.pop())
```

```
        elif x.text == "od":
            end_while_id = last_id
            (statement, last_id) = layer_stack.pop()
            graph.add_edge(end_while_id, last_id)
        print "id:",id," statement: ",program
        if(x.text != "fi" and x.text != "od"):
            s = ""
            if(len(un_solved_stack) != 0):
                s = un_solved_stack.pop()
            while(s != "then" and s != "else" and s != ""):
                graph.add_edge(s, id)
                    s = ""
                if(len(un_solved_stack) != 0):
                    s = un_solved_stack.pop()
            if(s == "then" or s == "else"):
                un_solved_stack.append(s)
        id += 1
        program = ""
    return graph
```

5 Data Structures for Reaching Definitions

Since python allows for some nice set notation, we decided to implement everything using python's built in sets. Each set is populated with tuples of (x, l) where $x \in$ variables(program) and $l \in$ labels(program).

5.1 Data Flow Equations

In each of our CFG's we annotate each node with a list of the variables being updated at that particular label. Since the grammar is relatively simple, there is only one assignment that can occur at the *statement* level. To generate the Reaching Definitions for a given program, we first calculate the genSets and killSets for any given statement as:

```
def getGenSets(cfg):

    nodes = cfg.nodes()
    labels = (node.name for node in nodes)
    variables = (node.attr["var"] for node in nodes)

    genSets = dict()
    for node, label, variable in zip(nodes, labels, variables):
        genSets[label] = {(variable, label)} if variable != '' else set()

    return genSets


def getKillSets(genSets):
```

```
    killSets = dict()

    for label, genSet in genSets.iteritems():
        killSets[label] = set()

        if genSet:
            for generation in genSet:
                variable = generation[0]

                for gs in genSets.values():
                    for v, l in gs:
                        if v == variable and l != label:
                            killSets[label] |= gs

    return killSets
```

After creating the kill and gen sets for an arbitrary statement, we can construct the reaching definitions with the following bit of code:

```
def getReachingDefinitions(cfg, startLabel):
    """
    Calculates the reaching definitions of a control-flow graph.

    :param AGraph cfg: The control-flow graph for which to calculate RDs.
    :param str startLabel: The label of the start node of the CFG.
    :rtype: dict[str, set[(str, str)]]
    """

    # Gen / Kill sets can be generated completely immediately.
    # IN / OUT start out empty initially.
    # The previousOut set is used to track changes to OUT sets during iteration.
    genSets = getGenSets(cfg)
    killSets = getKillSets(genSets)
    inSets = {label: set() for label in genSets}
    outSets = inSets.copy()
    previousOutSets = {label: None for label in genSets}

    # The iteration queue stores the next nodes to evaluate in the iteration.
    queue = deque([startLabel])

    while queue:
        label = queue.popleft()

        # Calculate (new) IN set.
        inSets[label] = getInSet(label, cfg, outSets)

        # Calculate the (new) OUT set, and check if it is different from the
```

```
        # previous version so we know whether or not to add the node's
        # successors to the iteration queue. The previous set is initially
        # populated by None, so every node will be visited at least once.
        outSet = getOutSet(label, genSets, killSets, inSets)
        if outSet != previousOutSets[label]:
            outSets[label] = outSet
            previousOutSets[label] = outSet
            queue.extend((s.name for s in cfg.successors(label) if s not in queue))

    return inSets
```

The function to calculate the reaching definitions for a particular entry is RD(block) = RD(block) − killSet(block) ∪ genSet(block). This is applied iteratively until there are no changes in the previously calculated RD for any given block. There are two utility functions (*getInSet* and *getOutSet*) that are not mentioned here, but can be found in the code appendix.

Code Appendix

5.2 AST To CFG Code

```
from pyparsing import *
from types import *
from copy import *
from pygraphviz import *


class node(object):
    def __init__(self, value, children = []):
        self.value = value
        self.children = children

    def __repr__(self, level=0):
        ret = "\t"*level+repr(self.value)+"\n"
        for child in self.children:
            ret += child.__repr__(level+1)
        return ret

    def delete_child(self, child):
        self.children.remove(child)

    def add(self,value):
        self.children.append(node(value))

def render(root):
    if root.value == '$' or root.value == 'then' or root.value == 'else' or root.value == 'do':
        process = root.children
```

```
    root.children = []
else:
    process = root.value
    root.value = ''
    root.children = []
if(type(process) is IntType or type(process) is StringType):
    root.value = process
elif(len(process) == 1):
    root.value = process[0]
else:
    list = []
    sub_list = []
    st = []
    for x in process:
        if x == 'if' or x == 'then' or x == 'else' or x == 'while' or x == 'do':
            sub_list.append(x)
            st.append(x)
        elif x == ";":
            if len(st) == 0 and len(sub_list) != 0:
                temp = copy(sub_list)
                list.append(temp)
                sub_list = []
                temp = []
            elif len(sub_list) != 0:
                sub_list.append(x)
        elif x == 'fi' or x == 'od':
            if x == 'fi':
                sub_list.append(x)
                st.pop()
                st.pop()
                st.pop()
            else:
                sub_list.append(x)
                st.pop()
                st.pop()
            if len(st) == 0:
                temp = copy(sub_list)
                list.append(temp)
                sub_list = []
                temp = []
        else:
            sub_list.append(x)
    if len(sub_list) != 0:
        temp = copy(sub_list)
        list.append(temp)
```

```
if(len(list) > 1 or root.value == '$' or root.value == 'then' or root.value == 'else' or root.va
    for x in list:
        root.children.append(node(x))
elif(list[0][0] == 'if'):
    root.value = 'if'
    list[0].remove('if')
    list2 = []
    sub_list = []
    st = []
    temp = []
    for x in list[0]:
        if (x == 'then' or x == 'else' or x == 'fi') and len(st) == 0:
            temp = copy(sub_list)
            list2.append(temp)
            temp = []
            sub_list = []
        elif x == 'fi':
            sub_list.append(x)
            st.pop()
            st.pop()
            st.pop()
        elif x == 'then' or x == 'else' or x == 'if':
            sub_list.append(x)
            st.append(x)
        else:
            sub_list.append(x)
    root.children.append(node(list2[0]))
    root.children.append(node('then', list2[1]))
    root.children.append(node('else', list2[2]))
elif(list[0][0] == 'while'):
    root.value = 'while'
    list[0].remove('while')
    list2 = []
    sub_list = []
    st = []
    temp = []
    for x in list[0]:
        if(x == 'do' or x == 'od') and len(st) == 0:
            temp = copy(sub_list)
            list2.append(temp)
            temp = []
            sub_list = []
        elif x == 'od':
            sub_list.append(x)
            st.pop()
            st.pop()
```

```
                elif x == 'while' or x == 'do':
                    sub_list.append(x)
                    st.append(x)
                else:
                    sub_list.append(x)
            root.children.append(node(list2[0]))
            root.children.append(node('do', list2[1]))
        elif(list[0][0] == 'skip'):
            root.value = 'skip'
        elif(len(list[0]) > 1):
            if(list[0][0] == 'not'):
                root.value = 'not'
                root.children.append(node(list[0][1:]))
            else:
                root.value = list[0][1]
                root.children.append(node(list[0][0]))
                root.children.append(node(list[0][2:]))

    for x in root.children:
        render(x)

def convert_to_graph(root):
    id = 1
    queue = []
    graph = AGraph()
    graph.add_node(0, label = root.value)
    queue.append((0,root))
    while(len(queue) != 0):
        (new_id,new_root) = queue.pop(0)
        if(len(new_root.children) != 0):
            for x in new_root.children:
                graph.add_node(id, label = x.value)
                graph.add_edge(new_id, id)
                queue.append((id, x))
                id += 1
    return graph

def to_string(root):
    if len(root.children) == 0:
        return str(root.value) + "\n";
    if root.value == ":=":
        return root.children[0].value + " := " + str(to_string(root.children[1])) + "\n"
    if root.value == "+" or root.value == "-" or root.value == "*" or root.value == "<" or root.value ==
        return str(root.children[0].value) + " " + root.value + " " + str(root.children[1].value) + "\n
    if root.value == "if":
        return "if " + str(to_string(root.children[0])) + "\n"
```

```
    if root.value == "not":
        return "not " + str(root.children[0].value) + "\n"
    if root.value == "while":
        return "if " + str(to_string(root.children[0])) + "\n"
def convert_to_cfg(root):
    id = 0
    graph = AGraph()
    program = ""
    revers_stack = []
    for x in root.children:
        revers_stack.append(x)
    stack = []
    while(len(revers_stack) != 0):
        stack.append(revers_stack.pop())
    layer_stack = []
    while(len(stack) != 0):
        print '--------------'
        print stack
        x = stack.pop()
        print x
        print layer_stack
        if x.value == 'if':
            temp = to_string(x)
            program += temp
            graph.add_node(id, label = program)
            if len(layer_stack) != 0:
                (temp,last_id) = layer_stack.pop()
                if temp == "endelse":
                    graph.add_edge(last_id, id)
                    (temp,last_id) = layer_stack.pop()
                    graph.add_edge(last_id, id)
                if temp == "if":
                    graph.add_edge(last_id, id)
                    layer_stack.append((temp,last_id))
            layer_stack.append(("if",id))
            program = ""
            id += 1
            stack.append(x.children[2])
            stack.append(x.children[1])
        elif x.value == "then" or x.value == "else":
            if x.value == "then":stack.append(node("endthen"))
            if x.value == "else":stack.append(node("endelse"))
            revers_stack = []
            for xx in x.children:
                revers_stack.append(xx)
            while(len(revers_stack) != 0):
```

```
                stack.append(revers_stack.pop())
        elif x.value == "endthen" or x.value == "endelse":
            print "program :"+program+str(id)
            print len(program)
            if len(program) != 0:
                graph.add_node(id, label = program)
            (temp, last_if_id) = layer_stack.pop()
            revers_stack = []
            while(temp != "if"):
                revers_stack.append((temp, last_if_id))
                (temp, last_if_id) = layer_stack.pop()
            while(len(revers_stack) != 0):
                layer_stack.append(revers_stack.pop())
            if len(program) != 0: graph.add_edge(last_if_id,id)
            if x.value == "endthen":
                layer_stack.append(("endthen",id))
                layer_stack.append((temp, last_if_id))

            if x.value == "endelse":
                (temp, last_then_id) = layer_stack.pop()
                #graph.add_edge(last_if_id, last_then_id)
                layer_stack.append(("endthen", last_then_id))
                layer_stack.append(("endelse", id))
            program = ""
            id += 1
        elif x.value == "while":
            temp = to_string(x)
            program += temp
        else:
            program += str(to_string(x))
    print layer_stack
    print graph
    return graph


boolean_op = oneOf('&& ||')
arith_bool_op = oneOf('== >= <= > <')
arith_op = oneOf('+ - *')
integer_value = Word(nums).setParseAction(lambda t:int(t[0]))
boolean_value = Keyword("true") | Keyword("false")
variable = Word( srange("[a-zA-Z_]"), srange("[a-zA-Z0-9_]") )
integer_variable = variable | integer_value
boolean_variable = variable | boolean_value


boolean_operation = integer_variable + arith_bool_op + integer_variable | boolean_variable + boolean_op
```

```
operation = Keyword("not") + boolean_variable | integer_variable + arith_bool_op + integer_variable | i

simple_stmt = Keyword("skip") | variable + ":=" + operation

stmt = Forward()
stmt << ((simple_stmt + ";" + stmt) | "while" + boolean_operation + "do" + stmt + "od" + Optional(";" +


parsed_list = stmt.parseFile("test.txt")
ast = node('$',parsed_list)
render(ast)
print ast


A = convert_to_graph(ast)

#print(A.string()) # print to screen
A.layout(prog='dot')
'''
print "writing to ast.dot"
A.write("ast.dot") # write to simple.dot
'''
print "printing to ast.png"
A.draw('ast.png',prog="dot") # draw to pngi using circo


cfg = convert_to_cfg(ast)
#print cfg.string()
cfg.layout(prog='dot')
print "printing to cfg.png"
cfg.draw('cfg.png',prog="dot") # draw to pngi using circo
```

```
from pygraphviz import *
from copy import *

'''return the rhs of an assignment to string'''
def render_short(node):
    temp_program = ""
    x = node.text
    if x == "+" or x == "-" or x == "*" or x == ">" or x == "<" or x == ">=" or x == "<=" or x == "&&"
        temp_program = str(node.children[0].text) + " " + x + " " + str(node.children[1].text)
    elif x == "not":
        temp_program = x + " " + str(node.children[0].text)
    else:
        temp_program = str(x)
    return temp_program
```

```
'''converting an AST to CFG'''
def convert_ast_to_cfg(ast):
    graph = AGraph()
    id = 0
    program = ""
    revers_stack = []
    for x in ast.children:
        revers_stack.append(x)
    stack = []
    while(len(revers_stack) != 0):
        stack.append(revers_stack.pop())
    layer_stack = []
    last_id = 0
    un_solved_stack = []
    while(len(stack) != 0):
        x = stack.pop()
        print x.text
        print un_solved_stack
        if x.text == ":=" or x.text == "skip":
            if x.text == "skip":
                program = "skip";
            else:
                program = str(x.children[0].text) + " := " + render_short(x.children[1])
            graph.add_node(id, l = program)
            if(last_id != 0):
                graph.add_edge(last_id, id)
            last_id = id
        elif x.text == "if":
            program = "if " + render_short(x.children[0])
            graph.add_node(id, l = program)
            if(last_id != 0):
                graph.add_edge(last_id, id)
            fi_node = copy(x.children[0])
            else_node = copy(x.children[2])
            then_node = copy(x.children[1])
            fi_node.token.text = "fi"
            else_node.token.text = "else"
            then_node.token.text = "then"
            stack.append(fi_node)
            stack.append(else_node)
            stack.append(then_node)
            layer_stack.append(("if", id))
            last_id = id
        elif x.text == "then":
            un_solved_stack.append("then")
```

```
        revers_stack = []
        for xx in x.children:
            revers_stack.append(xx)
        while(len(revers_stack) != 0):
            stack.append(revers_stack.pop())
    elif x.text == "else":
        un_solved_stack.pop()
        un_solved_stack.append(last_id)
        un_solved_stack.append("else")
        revers_stack = []
        for xx in x.children:
            revers_stack.append(xx)
        while(len(revers_stack) != 0):
            stack.append(revers_stack.pop())
        (statement, last_id) = layer_stack.pop()
        layer_stack.append((statement,last_id))
    elif x.text == "fi":
        un_solved_stack.pop()
        un_solved_stack.append(last_id)
        layer_stack.pop()
        last_id = 0
    elif x.text == "while":
        program = "while " + render_short(x.children[0])
        graph.add_node(id, l = program)
        if(last_id != 0):
            graph.add_edge(last_id,id)
        od_node = copy(x.children[0])
        do_node = copy(x.children[1])
        od_node.token.text = "od"
        do_node.token.text = "do"
        stack.append(od_node)
        stack.append(do_node)
        layer_stack.append(("while", id))
        last_id = id
    elif x.text == "do":
        revers_stack = []
        for xx in x.children:
            revers_stack.append(xx)
        while(len(revers_stack) != 0):
            stack.append(revers_stack.pop())
    elif x.text == "od":
        end_while_id = last_id
        (statement, last_id) = layer_stack.pop()
        graph.add_edge(end_while_id, last_id)
    print "id:",id," statement: ",program
    if(x.text != "fi" and x.text != "od"):
```

```
            s = ""
            if(len(un_solved_stack) != 0):
                s = un_solved_stack.pop()
            while(s != "then" and s != "else" and s != ""):
                graph.add_edge(s, id)
                    s = ""
                if(len(un_solved_stack) != 0):
                    s = un_solved_stack.pop()
            if(s == "then" or s == "else"):
                un_solved_stack.append(s)
        id += 1
        program = ""
    return graph
```

5.3 Reaching Definitions

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

""" A collection of functions for calculating reaching definitions. """

from pygraphviz import AGraph
from collections import deque


def getGenSets(cfg):
    """
    Calculates the gen sets for a labeled control-flow graph.

    The gen set is a dictionary with keys that map to the cfg's node labels. The
    values of this dictionary are sets of tuples. Each tuple in the set is of
    the form (variable name, label) where the variable name is a string and the
    label is an integer. Example:

        1 -> set((x, 1))
        2 -> set()
        3 -> set((y, 3), (z, 3))

    Due to the simple nature of our arithmetic grammar, these sets will always
    only contain one item because a single statement can not assign more than
    one variable. Also, the label number in the tuple will always be the key.

    :param AGraph cfg: The CFG for which to generate the gen set.
    :rtype: dict[str, set[(str, str)]]
    """
```

```
    # Get generators for the various attributes for convenience.
    nodes = cfg.nodes()
    labels = (node.name for node in nodes)
    variables = (node.attr["var"] for node in nodes)

    # Generate the genSet. For non assignment nodes, the value an empty set.
    genSets = dict()
    for node, label, variable in zip(nodes, labels, variables):
        genSets[label] = {(variable, label)} if variable != '' else set()

    return genSets


def getKillSets(genSets):
    """
    Calculates a kill set for control-flow graph, given its gen set.

    The kill set is a dictionary with keys that map to the cfg's node labels.
    The values of this dictionary are sets of tuples. Each tuple in the set is
    of the form (variable name, label) where the variable name is a string and
    the label is an integer. Example:

        1 -> set((x, 9))
        2 -> set()
        3 -> set((y, 2), (z, 7))

    :param dict[str, set[(str, str)]] genSets: Gen set of the CFG.
    :rtype: dict[str, set[(str, str)]]
    """

    killSets = dict()

    # We'll be iterating over the gen set instead of the CFG because all of the
    # information we need to construct the kill set is contained in the gen set.
    # The kill set for a basic block is the set of all other generations for the
    # same variable, which is conveniently stored in the gen set.
    for label, genSet in genSets.iteritems():
        killSets[label] = set()

        # A basic block's gen set must have at least 1 item to have a kill set.
        # For each generation in a basic blocks gen set, retrieve the variable
        # that is generated so that we can look for generations in other basic
        # blocks' gen sets that modify the same variable.
        # Reminder: g = (var, label), so g[0] = var
        if genSet:
            for generation in genSet:
```

```
                variable = generation[0]

                # For each generation in each gen set (except generations with
                # the same label as the gen set we are inspecting, as those are
                # it's own generations!), for any generation that modifies the
                # same variable as the outer gen set, add that generation to the
                # kill set.
                for gs in genSets.values():
                    for v, l in gs:
                        if v == variable and l != label:
                            killSets[label] |= gs

    return killSets


def getInSet(label, cfg, outSets):
    """
    Calculates the IN set for a basic block (identified by its label).

    The IN set of a basic block is the union of OUT sets of its predecessors.
    The IN set is also the Reaching Definition (RD) Set.

    :param str label: The node for which to calculate the in set.
    :param AGraph cfg: The control-flow graph the node belongs to.
    :param dict[str, set[(str, str)]] outSets: Set of all out sets for the cfg.
    :rtype: set[(str, str)]
    """
    return set().union(*(outSets[p.name] for p in cfg.predecessors(label)))


def getOutSet(label, genSets, killSets, inSets):
    """
    Calculates the OUT set for a basic block (identified by its label).

    :param str label: The node ofr which to calculate the out set.
    :param dict[str, set[(str, str)]] genSets: Gen sets of the cfg.
    :param dict[str, set[(str, str)]] killSets: Kill sets of the cfg.
    :param dict[str, set[(str, str)]] inSets: IN sets of the cfg.
    :rtype: set[(str, str)]
    """
    return genSets[label] | (inSets[label] - killSets[label])


def getReachingDefinitions(cfg, startLabel):
    """
    Calculates the reaching definitions of a control-flow graph.
```

```
    :param AGraph cfg: The control-flow graph for which to calculate RDs.
    :param str startLabel: The label of the start node of the CFG.
    :rtype: dict[str, set[(str, str)]]
    """

    # Gen / Kill sets can be generated completely immediately.
    # IN / OUT start out empty initially.
    # The previousOut set is used to track changes to OUT sets during iteration.
    genSets = getGenSets(cfg)
    killSets = getKillSets(genSets)
    inSets = {label: set() for label in genSets}
    outSets = inSets.copy()
    previousOutSets = {label: None for label in genSets}

    # The iteration queue stores the next nodes to evaluate in the iteration.
    queue = deque([startLabel])

    while queue:
        label = queue.popleft()

        # Calculate (new) IN set.
        inSets[label] = getInSet(label, cfg, outSets)

        # Calculate the (new) OUT set, and check if it is different from the
        # previous version so we know whether or not to add the node's
        # successors to the iteration queue. The previous set is initially
        # populated by None, so every node will be visited at least once.
        outSet = getOutSet(label, genSets, killSets, inSets)
        if outSet != previousOutSets[label]:
            outSets[label] = outSet
            previousOutSets[label] = outSet
            queue.extend((s.name for s in cfg.successors(label) if s not in queue))

    return inSets


if __name__ == "__main__":
    # Manually construct a CFG for testing purposes from the following code:
    #
    # x := 4;
    # y := 7;
    # while x < 10 do
    #     if y > 3 then
    #         y := x + 1;
    #         x := x + 1;
```

```
#     else
#         a := x + 1;
#         y := x + 2;
#     fi;
#     x := x + 1;
# od;
#
# This code was adapted from:
# https://engineering.purdue.edu/~milind/ece573/2011spring/ps8-sol.pdf
g = AGraph(directed=True)

# Nodes
g.add_node("start", label="START",              var='')
g.add_node("0",     label="x := 4;",            var='x')
g.add_node("1",     label="y := 7;",            var='y')
g.add_node("2",     label="while x < 10 do",    var='')
g.add_node("3",     label="if y > 3 then",      var='')
g.add_node("4",     label="y := x + 1;",        var='y')
g.add_node("5",     label="x := x + 1;",        var='x')
g.add_node("6",     label="a := x + 1;",        var='a')
g.add_node("7",     label="y := x + 2;",        var='y')
g.add_node("8",     label="x := x + 1;",        var='x')
g.add_node("end",   label="END",                var='')

# Edges
g.add_edge("start", "0")
g.add_edge("0", "1")
g.add_edge("1", "2")
g.add_edge("2", "3")
g.add_edge("2", "end")
g.add_edge("3", "4")
g.add_edge("3", "6")
g.add_edge("4", "5")
g.add_edge("5", "8")
g.add_edge("6", "7")
g.add_edge("7", "8")
g.add_edge("8", "2")

# Output a diagram of the control-flow graph.
# g.draw("output/input2.png", format="png", prog="dot")

rd = getReachingDefinitions(g, u"start")
labels = sorted([k for k in rd.keys()])
for label in labels:
    print label + ": ",
    for s in rd[label]:
```

```
            print "[" + s[0] + ", " + s[1] + "] ",
        print
```

5.4 AST Nodes

```
#!/usr/bin/env python

""" Various custom AST node types that contain LLVM code generation methods. """

import antlr3
import antlr3.tree
from llvm.core import Module, Constant, Type, Value, ICMPEnum

from simpleLexer import *

# GLOBALS (directly from tutorials)
g_llvm_builder = None  # Builder created any time a function is entered.
tp_int = Type.int()
tp_bool = Type.int(1)


class EmitNode(antlr3.tree.CommonTree):
    def __str__(self):
        return self.text

    def emit(self):
        print "TYPE " + str(self.type) + " UNIMPLEMENTED"

    def getScope(self):
        """
        This will return the nearest BLOCK's scope. This is useful for updating
        a blocks scope from assignments.
        :rtype: dict[str, Value]
        """
        return self.parent.getScope()

    def getClosure(self):
        """
        Only Block nodes have variable scopes. Default behavior is to just ask
        for the parents closure. The root of the AST will always be a BLOCK, so
        it should always return from there.
        :rtype: dict[str, Value]
        """
        return self.parent.getClosure()
```

```
class BlockNode(EmitNode):
    def __init__(self, payload):
        super(BlockNode, self).__init__(payload)

        self.scope = {}
        """ The local scope for this block. """

        self.closure = None
        """ The closure of variables for this block. """

    def __str__(self):
        return '\n'.join([str(node) for node in self.children])

    def emit(self):
        for child in self.children:
            child.emit()
            # NOTE: Blocks do not have a return value!

    def initClosure(self):
        self.closure = []
        if self.parent is not None:
            # Use list() to create a new list rather than updating the parent's.
            self.closure = list(self.parent.getClosure())

        self.closure.append(self.scope)

    def getScope(self):
        """
        Returns this node's scope.
        :rtype: dict[str, llvm.core.PointerType]
        """
        return self.scope

    def getClosure(self):
        """
        Returns this node's closure. If the closure hasn't been initialized, do
        so first, then return it. This must be done here rather than in the
        constructor because parent/children data are not populated until AFTER
        the nodes have all been created.
        :rtype: dict[str, llvm.core.PointerType]
        """
        if self.closure is None:
            self.initClosure()
        return self.closure
```

```
class SkipNode(EmitNode):
    def __str__(self):
        return "SKIP"

    def emit(self):
        # Emit useless instruction as a no-op.
        zero = Constant.int(tp_int, 0)
        return g_llvm_builder.add(zero, zero, "noop")


class IntegerNode(EmitNode):
    def __str__(self):
        return self.text

    def emit(self):
        return Constant.int(tp_int, self.text)


class IdentifierNode(EmitNode):
    def __str__(self):
        return self.text

    def emit(self):
        name = self.text
        scope = [scope for scope in self.getClosure() if name in scope]
        if scope:
            scope = scope[0]
            return g_llvm_builder.load(scope[name], name)
        else:
            raise RuntimeError("Unknown variable name: " + self.text)


class UnaryNode(EmitNode):
    def __str__(self):
        return "".join([str(child) for child in self.children])

    def emit(self):
        op = self.children[0]
        expr = self.children[1].emit()
        if op.text == '-':
            return g_llvm_builder.neg(expr, "negated_" + expr.name)
        else:
            return expr


class BooleanNode(EmitNode):
```

```
    def __str__(self):
        return self.text

    def emit(self):
        if self.text.lower() == "true":
            return Constant.int(tp_bool, 1)
        elif self.text.lower() == "false":
            return Constant.int(tp_bool, 0)
        else:
            raise RuntimeError("Invalid boolean value.")


class UnaryBoolOpNode(EmitNode):
    def emit(self):
        b = self.children[0].emit()
        return g_llvm_builder.not_(b, "notbool")


class BinaryBoolOpNode(EmitNode):
    def emit(self):
        left = self.children[0].emit()
        right = self.children[1].emit()

        if self.text.lower() == '&':
            return g_llvm_builder.and_(left, right, "and")
        elif self.text.lower() == '|':
            return g_llvm_builder.or_(left, right, "or")
        else:
            raise RuntimeError("Unrecognized binary boolean operator.")


class AssignmentNode(EmitNode):
    def emit(self):
        name = self.children[0].text
        val = self.children[1].emit()

        # Find scope this variable exists in (if it exists already). If not,
        #  add the variable name to the current scope and allocate it.
        scope = [scope for scope in self.getClosure() if name in scope]
        if scope:
            scope = scope[0]  # Remove outer list generated by list comp.
        else:
            scope = self.getScope()
            scope[name] = g_llvm_builder.alloca(tp_int, name=name)

        # Update the value of the variable.
```

```
        return g_llvm_builder.store(val, scope[name])


class ArithmeticNode(EmitNode):
    def emit(self):
        left = self.children[0].emit()
        right = self.children[1].emit()

        if self.text == '*':
            return g_llvm_builder.mul(left, right, 'mul')
        elif self.text == '+':
            return g_llvm_builder.add(left, right, 'add')
        elif self.text == '-':
            return g_llvm_builder.sub(left, right, 'sub')
        else:
            raise RuntimeError("Unrecognized arithmetic operator.")


class RelationalNode(EmitNode):
    def emit(self):
        left = self.children[0].emit()
        right = self.children[1].emit()

        if self.text == '=':
            return g_llvm_builder.icmp(ICMPEnum.ICMP_EQ, left, right, "comp_eq")
        elif self.text == '<':
            return g_llvm_builder.icmp(ICMPEnum.ICMP_SLT, left, right, "comp_slt")
        elif self.text == '<=':
            return g_llvm_builder.icmp(ICMPEnum.ICMP_SLE, left, right, "comp_sle")
        elif self.text == '>':
            return g_llvm_builder.icmp(ICMPEnum.ICMP_SGT, left, right, "comp_sgt")
        elif self.text == '>=':
            return g_llvm_builder.icmp(ICMPEnum.ICMP_SGE, left, right, "comp_sge")
        else:
            raise RuntimeError("Unrecognized relational operator.")


class IfElseThenNode(EmitNode):
    def emit(self):
        conditional = self.children[0].emit()
        then_branch = self.children[1]
        else_branch = self.children[2]

        # Create blocks for the if/then cases.
        function = g_llvm_builder.basic_block.function
        then_block = function.append_basic_block('then')
```

```
        else_block = function.append_basic_block('else')
        continue_block = function.append_basic_block('continue')

        # Emit conditional instruction.
        g_llvm_builder.cbranch(conditional, then_block, else_block)

        # Emit then block contents.
        g_llvm_builder.position_at_end(then_block)
        then_branch.emit()
        g_llvm_builder.branch(continue_block)

        # Emit else block contents.
        g_llvm_builder.position_at_end(else_block)
        else_branch.emit()
        g_llvm_builder.branch(continue_block)

        # Place the builder in the continue block so that the rest of the tree
        # can be generated.
        g_llvm_builder.position_at_end(continue_block)


class WhileNode(EmitNode):
    def emit(self):
        loop = self.children[1]

        # Create blocks for the if/then cases.
        function = g_llvm_builder.basic_block.function
        condition_block = function.append_basic_block('while_condition')
        loop_block = function.append_basic_block('while_loop')
        continue_block = function.append_basic_block('continue')

        # Emit unconditional jump into while loop conditional block.
        g_llvm_builder.branch(condition_block)

        # Check conditional, then either branch into loop or out of loop.
        g_llvm_builder.position_at_end(condition_block)
        conditional = self.children[0].emit()
        g_llvm_builder.cbranch(conditional, loop_block, continue_block)

        # Emit loop block contents.
        g_llvm_builder.position_at_end(loop_block)
        loop.emit()

        # Emit unconditional branch back to conditional block so another check
        # can be performed.
        g_llvm_builder.branch(condition_block)
```

```
        # Place the builder in the continue block so that the rest of the tree
        # can be generated.
        g_llvm_builder.position_at_end(continue_block)


class ErrorNode(antlr3.tree.CommonErrorNode):
    def emit(self):
        raise RuntimeError("Could not parse input:\n\n    " + self.getText())


class LlvmAdaptor(antlr3.tree.CommonTreeAdaptor):
    def createWithPayload(self, payload):
        if payload is None:
            return EmitNode(payload)

        # Fixes the problem where ANTLR3 produces Unicode objects instead of
        # Python string objects, and LLVM does not like Unicode objects.
        payload.text = str(payload.text)

        t = payload.type
        if t == BLOCK:
            return BlockNode(payload)
        if t == SKIP:
            return SkipNode(payload)
        elif t == INTEGER:
            return IntegerNode(payload)
        elif t == IDENT:
            return IdentifierNode(payload)
        elif t == UNARY:
            return UnaryNode(payload)
        elif t in (MULT, PLUS, MINUS):
            return ArithmeticNode(payload)
        elif t == RELOP:
            return RelationalNode(payload)
        elif t == BOOLEAN:
            return BooleanNode(payload)
        elif t == NOT:
            return UnaryBoolOpNode(payload)
        elif t in (AND, OR):
            return BinaryBoolOpNode(payload)
        elif t == GETS:
            return AssignmentNode(payload)
        elif t == IF:
            return IfElseThenNode(payload)
        elif t == WHILE:
```

```
            return WhileNode(payload)
        else:
            return EmitNode(payload)

    def errorNode(self, input, start, stop, exc):
        return ErrorNode(input, start, stop, exc)
```

5.5 ANLTR Generated Code

Please see the project github if you wish to see this, it's too large an unruly to include in the report.