Classify Target Architecture

0 operand = “static machine”

all operands are on the stack

no need to store registers

ex. c = a+b

load a → load b → add → store c

1 operand = “accumulator machine”

the accumulator is an implicit parameter (will apply every time)

needs some space to store accumulator

ex. c = a+b

load a to accu → add b to accu → store c from accu

2 operand = “CISC”

ex. c = a+b

load a to register 1 → add b to register 1 → store register 1 to c

3 operand = “RISC”

ex. c = a+b

add register C ← register A, register B

## Simple Compiler

DLX = architecture without too many instructure formats

Ex.

context-free language → parser (push down automaton):

E → T {“+” T}

T → F {“\*” F}

F → “(”E“)” | number | ident

regular language → tokenizer (finite state machine):

number → digit {digit}

ident → letter {letter | digits}

digit → “0”|...|”9”

letter → “a”|...|”z”

1+2\*(x+y)

First version

void E {

T();

while (symbol == plustoken) {

next();

T();

emit(ADD);

};

};

void T() {

F();

while(symbol == multoken){

next();

F();

emit(MUL);

}

};

void F(){

if (symbol == openP){

next();

E();

if (symbol == closeP) next

else {syntaxError()}

} else if (number) {

emit(LOAD tokenizer.val);

next();

}

else if (identifier) {

emit(LOAD tokenizer.identifier);

next();

}

else syntaxError();

}

Instruction format (can be abstracted and put in mem for later calculation)

Mnemonic Operation Format Opcode

ADD a, b, c R.a := R.b + R.c F 2 0

If we want R6 ← R2 + R3, we have F2 as 00000 | 00110(6) | 00010(2) | 0\*11 | 00011(3) 

code array buf[]

counter int pc

Better Version

void putF1(){

int op, a, b, c

buf[pc] = op << 26 | #5+5+11+5

a << 21 | #5+11+5

b << 16 | #11+5

c & oxFFFF #only take the last 5 bits

}

void putF2() {}

void putF3() {}

tokenizer {

int id

int number

int getNext() {} #sets id or number

}

parser {

int symbol # token at front

void next() { symbol = tokenizer.getNext()}

checkFor( int expectedToken) {

if symbol == expectedToken {

next()

} else {

syntaxError()

}

}

void E() {

T()

while symbol == “+” {

next()

T()

PutF2(ADD 0, sp-1, sp-1, sp--)

}

}

void T() {... PutF2(MUL 2, sp-1, sp-1, sp) }

void F() {

if symbol == “(“ {

next()

E()

next()

}

elif symbol == number {

PutF1(ADDI 16, ++sp, 0, tokenizer.number)

next()

} elif symbol == letter {

PutF1(LDW 32, ++sp, BASE, lookup(tokenizer.id))

next()

}

}

}

⇒ many repetitive tasks

a+4\*c+d

ETF → return

LDW R1 ← mem(BASE + a.add)

TF → return

ADDI R2 ← R0 + 4

F → return

LDW R3 ← mem(BASE + c.add) sp = 3

MUL R2 ← R2 \* R3 sp = 2

ADD R1 ← R1 + R2 sp = 1

LDW R2 ← mem[Base + b.add) sp = 2

ADD R1 ← R1 + R2 sp = 1

a+4\*c+1

ETF → return

LDW R1 ← mem(BASE + a.add)

TF → return

ADDI R2 ← R0 + 4

F → return

LDW R3 ← mem(BASE + c.add) sp = 3

MUL R2 ← R2 \* R3 sp = 2

ADD R1 ← R1 + R2 sp = 1

ADDI R2 ← R0 + 1

ADD R1 ← R1 + R2

## A Better Compiler

with descriptor knowing its lower tree (more complex architecture needs more memory)

class Result {

int kind; // const or var or register

int val; // const

int add; // var

int regno; //register

}

int AllocateReg(); //return idle register number

DeallocateReg(int reg); //free register

Result E(){

Result x,y;

x = T()

while (symbol == “+”) {

next();

y = T()

**compute**(ADD, x, y) //x = x + y

}

return x

}

Result T() {

Result x,y

x = F()

while (symbol = “\*”) {

next()

y = F()

**compute**(MUL, x, y) //x = x \* y

}

return x

}

Result F() {

Result x

if symbol == “(“ {

next()

x = E()

if symbol == “)” return x

else syntaxError()

}

else {

x = new(Result)

if symbol == number {

x.kind = const

x.val = tokenizer.val

} elif symbol == id {

x.kind = var

x.add = lookup(tokenizer.id)

}

return x

}

}

void Compute(int op, Result x, Result y) {

if x.kind == const && y.kind == const {

if op == MUL x.val \*= y.val

elif op == ADD x.val += y.val

} else {

load(x)

if y.kind == const {

PutF2(immOP[op], x.regno, x.regno, y.val)

} else {

load(y)

PutF2(op, x.regno, x.regno, y.regno)

DeallocateReg(y.regno)

}

}

}

const var reg

const const arithmetic

var

reg directly add normal reg-operation

void load(Result x) { //move x to reg ⇒ x.kind = reg

if x.kind == const {

if x.val == 0 {

x.regno = 0

x.kind = reg

} else {

x.regno = AllocateReg()

PutF2(ADDI 16, x.regno, 0, x.val)

x.kind = reg

}

}elif x.kind == var {

x.regno = AllocateReg()

PutF2(LDW 32, x.regno, BASE, x.add)

x.kind = reg

}

}

a + b \* 2 + 1

ETF(var add) return x back to E →

TT(var add) return x back to T →

T(const 2) return y back to T →

load x to reg1 LRW R1 ← BASE + b’adr

ADDI R1 ← R1 + 2

return y back to E and combine →

load x to reg2 LRD R2 ← BASE + a’adr

ADD R2 ← R2 + R1

free reg1 R1 free

Main Project: instead of computing the result, build the whole tree and compute

Result E(){

Result x,y;

x = T()

while (symbol == “+”) {

next();

y = T()

**build**(ADD, x, y) //x = x + y

}

return x

}

Result T() {

Result x,y

x = F()

while (symbol = “\*”) {

next()

y = F()

**build**(MUL, x, y) //x = x \* y

}

return x

}

[instruction no | op | instruction pointer | instruction pointer]

[instruction no | const | val]

expression: modify the variable table

if else: update variable table

while:

## SSA (Static Single-Assignment)

Control Flow Graph CFG

basic block = sequence of instructions that executed atomically

if

/ \

**full through branch**

/ \

then else

\ /

**branch full through**

\ /

follow (phi x\_left, x\_right)

* if and else should be balanced ⇒ attacker may guess the program by timing how long it takes
* reducible if it can be partitioned into two disjoint sets

the forward edge set which forms a directed acyclic graph and

the back edge set in which every head dominates its tail

“Dominator” relationship = block D dominates block B if every path from start to B includes D

ex.

B1

B2 B3 B4

“Post Dominator” = block P post dominates block B if every path from B to end includes P

ex.

B4

B1 B2 B3

redundant computation ⇐ same operation + same opprands + block(i) dominates block(j) (every path to j goes through i)

ex. i : op x y … j:op x y

Same operand: every assignment creates a new variable (same name ⇒ same value)

⇒ holds many different names for the same original variable

special operation to combine different incarnations of the same original variable

def = every place where different instnation of the same original variable flow together combine them using phi(x,y)

left branch =i+1 right branch = i - 1

merging branch i3 =

* phi’s are constraints for the register allocators

IR (Intermediate Representation??)

[instruction no | op | instruction pointer | instruction pointer]

[instruction no | const | val]

instruction = SSA value (for same operand assignment)

table for program variable → SSA value

Ex.

IR E() {

IR x,y

x = T()

while symbol == “+” {

next()

y = T()

x = build(“+”, x,y)

}

return x

}

### IF Statement

Ex. I = instruction, R = Register, op(operation), ip(instruction pointer)

a ← read

b ← a

c ← b

d ← b+c

e ← a+b

if a < 0 {

a ← d+e

} else {

d ← e + 1

}

write(a)

write(d)

BB0

|

BB1

| \

BB2 BB3 (branching block)

| /

BB4 (join block)

BBO for const I# ip

* BB points to Instruction and I# is only for debugging

I# op val

3 const #0

9 const #1

BB1

* add points to I2 and I2 points to None (⇐ no previous dominated add)
* candidate I for e: add 1 1 exists + I2 has no dominating I ⇒ use existing I
* When branching, the operation pointer table needs copying to all branching blocks and join block (how to go back and update the branching? should I finish the branching blocks before creating any phi?)

| | I# | op | ip | ip | | --- | --- | --- | --- | | 1 | read() |  |  | | 2 | add | 1 | 1 | | 4 | cmp | 1 | 3 | | 5 | bge | 10 |  | | | variable | I# | | --- | --- | | a | 1 | | b | 1 | | c | 1 | | d | 2 | | e | 2 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

BB2

* For I6, the add (in the copied operator table) points to I6 and I6 points to I2 as dominating I
* updating a to 6 in the newly copied table + no change in the other branch ⇒ phi function for a
* every assignment modifies a phi function in the current join block. If no phi function exists already, create one
* create phi function for a in BB4
* For the copied table, do we maintain all values? or just the modified values? same question for join table

| new copied table?   | I# | op | ip | ip | | --- | --- | --- | --- | | 6 | add | 2 | 2 | | 8 | bra | 7 |  | | new copied table   | variable | I# | | --- | --- | | a | 6 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

BB3

* add points to I10 and I10 points to I2

| new copied table?   | I# | op | ip | ip | | --- | --- | --- | --- | | 10 | add | 2 | 9 | |  |  |  |  | | new copied table   | variable | I# | | --- | --- | | e | 10 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

BB4

| new copied table?   | I# | op | ip | ip | | --- | --- | --- | --- | | 7 | phi | 6 | 1 | | 11 | phi | 2 | 10 | | 12 | write | 7 |  | | 13 | write | 11 |  | | new copied table   | variable | I# | | --- | --- | | a | 7 | | b |  | | c |  | | d |  | | e | 11 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Register Allocator

* For the phi functions, we want all of them to be assigned to the same place by Register Allocator. 7, 6, 1 are in R1 and 11, 2, 10 are in R2

1 Read R1

2 Add R1, R2 → R2

4 CMP R1, #0

5 BGE 10

6 ADD R1, R2 → R1

8 BRA 12

10 ADD R2, #1 → R2

12 print R1

13 print R2

* If we are copying the variable I# table by making assignment ⇒ this table is not renamed by the phi. It lives across the phi ⇒ we need more memory to make the copy at the phi function or ahead of time (before branching)

Ex. Ask

a ← read

if a > 0 {

a = a + 1

}

print(a)

| I# | operator | instruction pointer | ip |
| --- | --- | --- | --- |
| 1 | read |  |  |
| 2 | const | #0 |  |
| 3 | cmp | 1 | 2 |
| 4 | ble | 8 |  |
| 5 | const | #1 |  |
| 6 | add | 1 | 5 |
| 7 | bra | 8 |  |
| 8 | phi | 6 | 1 |
| 9 | write | 8 |  |

Ex. I = instruction, R = Register, op(operation), ip(instruction pointer)

Instruction #i op ip ip

a ← read I 1 read

b ← a

c ← b a,b,c all points to the same memory

d ← b+c I 2 add 1 1

e ← a+b d,e points to the the same memory

if a < 0 { I 3 cmp 1 4

I 4 const #0

I 5 bye?? …

d ← d+e I 6 add 2 2

a ← d I 7 branch()

} else {

d ← e + 1 I 8 add 3 9

I 9 const #1

}

I 10 phi (6)(8) for d

I 11 phi (6)(1) for a

print(a) I 12 write 11

I and basic blocks

I 1 2 3 5 are in the same basic block

I 4 is in the initial basic block

I 6 7 are in same basic block

I and R

1 read a

2 add R1, R1 → R2

3 comp R1 #0

5 branch equally(bye?) 8

—

6 add R2, R2 → R1

7 branch 12

—

8 add R1 #1 → R2

—

12 write R1

* use as few registers as possible to save execution time ⇒ decide what variables are more frequently used (ex. inside a loop)

### WHILE Statement

BB0

|

BB1 ←— |

| \ |

BB3 BB2

* BB2 is the loop body, BB3 is the follow
* BB1 has phi function for the loop body and previous block

Ex.

i ← read ()

x ← 0

y ← 0

j ← i

while x < 10 do

x ← i + 1

y ← j + 1

i ← i + 1

od;

BB0

I# op val

2 const #0

3 const #10

6 const #1

BB1

| | I# | op | ip | ip | | --- | --- | --- | --- | | 1 | read() |  |  | | | variable | I# | | --- | --- | | i | 1 | | x | 2 | | y | 2 | | j | 1 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

BB2

* Before generating I4, leave space for several phi functions at the beginning of the loop
* phi cannot be reduced

| | I# | op | ip | ip | | --- | --- | --- | --- | | … | … | … | … | | 4 | cmp | 2 → 8 | 3 | | 5 | bge |  |  |   later generated phi functions   | 8 | phi | 2 | 7 | | --- | --- | --- | --- | | 9 | phi | 2 | 7 | | 10 | phi | 1 | 7 | | | Variable | I# | | --- | --- | | x | 8 | | y | 9 | | i | 10 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

BB3

* In I7, we generate a phi function I8 for x and add this phi to BB2 ⇒ we need to change all occurrence of x in after I8 to be I8

| first round   | I# | op | ip | ip | | --- | --- | --- | --- | | 7 | add | 1 | 6 | |  |  |  |  | | | variable | I# | | --- | --- | | i | ??? | | x | ??? | | y | ??? | | j | 1 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

* i is changed but j is not ⇒ x and y has different value ⇒ cannot be reduced

| second round   | I# | op | ip | ip | | --- | --- | --- | --- | | 7 | add | 10 | 6 | | 11 | add | 1 | 6 |   phi function in BB2 also changes??   | 9 | phi | 2 | 11 | | --- | --- | --- | --- | | | variable | I# | | --- | --- | | i | 7 | | x | 7 | | y | 11 | | j | 1 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Ex.

i ← read ()

x ← 0

y ← 0

j ← i

while x < 10 do

x ← i + 1

y ← j + 1

i ← i + 1;

od;

* every assignment modifier or creaks a phi function in the current join block

BB0 contains all the constants

2. constant #0 [x as (2), y as (2), j as (1)]

3. consta #10

6. const #1

BB1

1. read [ i as (1)]

BB2 loop header

/ \bra ←—

body bra? follow |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

8. phi (2) (7) [x as (8)] ⇒ (4) should be updated to cmp (8)

10. phi (2) (7) [y as (10)]

11. phi (1) (7) [i as (11)]

4. cmp (2) ⇒ (8)

5. bge

follow

…

body bra

7. add (1) (6) [x as (7), y as (7), i as (7)]

12. add (1) (6) [i as

…

Ex.

max(a[N, N])

do i ←1 to N

let large(i) ← 1

large(

Main Project

ident = letter {letter | digit}.

varDecl = “**var**” | “**array**” indent { “,” ident } “;”

funcDecl = [ “void” ] “function” ident formalParam “;” funcBody “;” .

computation = “main” { varDecl}

{ funcDecl }

**“{”** statSequence “}” “.” .

if “var” or “array” ⇒ varDecl

if “void” or “function” ⇒ funcDecl

⇒ statSequence

### Array

SSA model only applies to data that can be more efficiently

indexed load

indexed store

adda

load

adda

store

Ex.

i = read()

j = read()

x = a[i]+ 2

3 const #size\_of\_element\_in\_array

5 #a\_base\_address

9 const #2

1 read() ← i

2 read() ← j

4 mul (1) (4)

6 add BASE (5)

[

7 adda (4) (6)

8 load (7)

]

10 add (8) (9) ← x\

Ex.

Ex. a[i] = x + y

3 const #size\_of\_element\_in\_array

5 const #a’adr

42 add x y

43 mul (i) (3) can change

44 add base (5) ← pointer to array this does NOT change

45 adda 43 44 ← a[i] can change

46 store 45 42

Ex.

BB1

i = read()

j = read()

a[i] = x+y

print a[i]...

/ \

BB2

print a[i]

a[j]=?

print a[i]

\ /

print a[i]

* if i != j, then a[j] doesn’t change. If i = j
* “kill” instruction needs to adding to the join block because data may be partially unavailable when a store happens in a non-dominating block

0 const #0

18 const #a

20 const a’adress

BB1

1 read

2 read

…

17 add x y

19 mul 1 18

21 adda 19 21

23 store 22 17

24 adda 19 21

25 load 24

26 write 25

cmp

bra

* **load operator should also search for store + load in the search table ⇒ 24&25 is unnecessary because load the same value as 23 ⇒ can directly write 23**

BB2

27 write 25

28 mul 2 18

29 adda 28 21

30 store 29 0

32 adda 19 21

33 load 32

34 write 33

BB3 – join

31 store ..a (insert to the search table??)

* store and load should points to I31 and I31 points to load/store in BB1

### Alias Analysis

“may alias” always some paths through program

“must alias”

low precision high precision

|\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_|

efficient flow sensitive costly

Ex.

int a, b, c[100], d, i, extein, int \*q

…

q = &a # points to a

a = z

b = \*2 + 2

…

q = &b # points to b

for i in range(100):

c[i] = c[i] + a

\*q = i

d = \*q + a

ALAT

## Search Data Structure

organized by opcode in while all previous instructions are guaranteed to dominate the current place

implemented with linked list (pointing to the last seen instruction that dominates)

ex. A if … then B else C endif D

A dominates all the following. If A has a1, a2 with the same opcode.

After A, opcode → a2 → a1

Before going inside B, make a copy of A for later C and D to use

B has b1, b2. After B, opcode → b2 → b1 → a2 → a1

C has c1, c2. After C, opcode → c2 → c1 → a2 → a1

…

SSA → Register Allocator

ex.

x ← k+ 1 x ← b-1

\ /

phi x (might cause issue when x has a copy (ex. t ← x in left branch) )

## Video 1 Tokenizer & Parser

LL1 = LL(1) grammars that can be parsed reading from Left to right, while forming the Leftmost derivation of the parse tree, using a 1-symbol lookahead

Three-Tiered Read-Ahead Architecture = characters ⇒ tokens ⇒ parser

getting characters

public class FileReader; {

public char GetNext(); // return current **character** and go next

public void Error(String errorMsg); // signal an error message

public FileReader(String fileName); // constructor: open file

}

characters → tokenizer (parser for a regular grammar)

public class Tokenizer; {

private FileReader myFileReader;

private char inputSym; // the current character on the input

private void next(); { inputSym = myFileReader.GetNext(); } // advance to the next character

/\* symmetrical to the FileReader class \*/

public int GetNext(); // return current **token** and go next

public int **number**; // the last number encountered

public int **id**; // the last identifier encountered ()

* value is stored in a table for lookup
* for the reserved identifier (if, else…) the value is also stored in the table to avoid duplicate

public void Error(String errorMsg); { myFileReader.Error(errorMsg); ... }

public Tokenizer(String fileName); // constructor: open file and cache its frontmost char in “inputSym’

/\* identifier table methods \*/

public String Id2String(int id);

public int String2Id(String name);

...

}

token → parser (build a parse tree. number: value; ident: id)

public class Parser; {

private Tokenizer myTokenizer;

private int inputSym; // the current token on the input

private void next(); { inputSym = myTokenizer.GetNext(); } // advance to the next token

void CheckFor(int token); {if (inputSym==token) then next(); else myTokenizer.Error(“SyntaxErr”); }

...

public Parser(String fileName); // constructor: open file and cache its frontmost token in “inputSym’

{

myTokenizer = new Tokenizer(filename); next();

CheckFor(mainToken);

...

CheckFor(periodToken);

}

...

}

Ex. A = Assignment, E = Expression, T=Term, F=Factor

S → [“int” id {“,” id} “;”] A {“;” A}.

A → “let” id “ ← ” E

E → T {“+” T}

T → F {“\*” F}

F → “(”E“)” | number | id

number → digit{digit}

digit → “0”|...|“9”

id → letter{letter|digit}

letter → “a”|...|“z”

* reserved: int let + \* ( ) , ; .
* “let” will avoid the case where we are parsing letters and not knowing if it’s a function or a variable assignment

Implementation:

Tokenizer

getNext(){

while (symbol == “ ”) {next();}

if symbol == “eof” return EofToken

else:

switch (symbol):

case “\*”:

next();

return TimesToken

case number:

number()

return NumberToken

…

}

number() {

val = int(symbol)

next()

while (symbol <= “9” and symbol >= “0”) {

val = val \* 10 + int(symbol)

next()

}

}

Parser

checkFor (int expectedToken) {

if (symbol == expectedToken) {next}

else {syntaxError()}

}

void S() {

if symbol == “int” {

next();

checkFor(intToken);

processDeclaration(Tokenizer.id)

while (symbol == “,”) {

next()

checkFor(identifierToken)

processDeclaration(Tokenizer.id)

}

checkFor(SemicolonToken)

}

…

}

## Video 2 LL Grammar & Parser

Context-free grammar + push-down automation

Two ways to parse sentence

LL parser (top-down or goal-oriented)

LL1 = from L, forming L-most derivation, 1 symbol look ahead

LL1 = subset(LL2), LL2 = subset(LLn), LLn = subset(LR)

top-down = apply the rule until you find the sentence (may use backtracking)

LR parser = bottom-up or shift-reduce = from the sentence, apply the inverse of the rule (usually more difficult)

bottom-up parser for c = YAC (Yet Another Compiler) + Lex (Alexa generator)

Ex. S → “(” S “)” | “x” produces (((x)))

Top-down S → (S) → ((S)) → (((S))) → (((x)))

Bottom-up (((x))) → (((S))) → ((S)) → (S) → S

LL1 Properties

For

For (can be reduced to empty set ⇒ optional ⇒ disjoint from ) and

If is nullable, id can be produced to the empty sentence, then,

For [exp] and {exp},

= set of all terminal symbols that can come right after X in any sentences of the language.

If X can come at the end, then follow includes the the end mark

* is the set of all terminals that can start sentences derived from X

Ex. first follow

E → T {“+” T} first(T) = “(” or “x” “)” or end mark

T → F {“\*” F} first(F) = “(” or “x” “+” or “)” or end mark

F → “(”E“)” | “x” “(” or “x” “\*” or “+” or “)” or end mark

Ex. counterexample

S → “if(“ E “)” S | “if(“ E “)” S “else” S | “a”

E → “b”

is the same as

S → “if(“ E “)” S S’| “a”

S’ → “else” S | “”

E → “b”

first(S’) includes “e” and follow(S’) includes “e” ⇒ ambiguity

problem = we don’t know if the else belongs to inner if or outer if

## Register Allocator

Background

before SSA: variable ⇔ storage location

after SSA: value ⇔ storage location (totally detached from variable names

model: infinite register set but more register ⇒ expensive

Interference Graph

Vertex: SSA value

edge between node x and node y = their lifetime overlaps

graph coloring (no adjacent nodes are colored the same) NP Hard Problem

Data structure

IR links to 0…\* IRs

keep track of live set (IR set)

Ex. code → SSA → interference graph → register optimization

a ← read()

b ← read()

write(a\*b+a\*7)

4 const #7

value (1) value (2) value (3) value (5) value (6)

1 read [start] {}

2 read | [start] {1}

3 mul (1)(2) use use[end] [start] {1,2}

5 mul (1)(4) use[end] | [start] {1,3}

6 add (3)(5) use[end] use[end] [start] {3,5}

7 write (6) use[end] {6}

⇒ we have the following interference graph and start coloring 2 with R1…

1 (R2) — 2 (R1)

|

3 (R1) — 5 (R2) 6

⇒ we can get the following

1 → R2

2 → R1

3 → R1

5 → R2

6 → R1

* in each BB from the bottom, we look at the live set and get the overlapping nodes
* if a value has no uses, we don’t need to compile it

Ex. If Else

BB1

| \

BB2 BB3

| /

BB4

BB4 → i: phi (x, y) and j: phi (t, u)

⇒ BB2 has live 1 - {i, j} + {x, t} at the end and live 2 at the start

⇒ BB3 has live 1 - {i, j} + {y,u} at the end and live 3 at the start

⇒ BB1 → live 2 + live 3

Ex. Loop

|(x,t)

BB1 ←—

| \ |

BB3 BB2

BB3 (follow) has live 1

BB1 → i: phi (x, y) and j: phi (t, u)

⇒ BB2 at the end has live 1 + {y, u}, at the start has live 2

⇒ BB1 at the end has live 1 and live 2, at the start has live 3

⇒ BB1 before the start has live 3 + {x, t}

Ex. nested loop (t interferences with all the inner loops and its returned outer loop)



Ex. coloring

void color(graph g):

x ← node with the lowest cost (from the boundary of the graph)

remove x from g

if g not empty:

color(g)

add x and its edge back to g

choose color for x that is different from its neighbors

cost function = # uses scaled by nesting depth

* “coalesce” = if argument/result of a phi do not interfere with each other, cluster them

Ex. coloring with phi

for all phi instructions “x=phi (y1, y2…yi)

cluster = {x}

for all yi that are not constant

if yi does not interfere u/ cluster

add yi and its edge to clusters.

remove yi from graph

Ex. code → SSA → interference graph

let a ←input

let b ← a

c ← b

d ← b+c

e← a+b

y ← a

if a < 0 then

d ← d+e

a ← d

else

d ← e

fi

output(a)

output(y)

output(d)

SSA



Interference graph



Clustering 762(R3) 1(R1) 8(R2)

1 read → R1

2 add R1, R1 → R3

…

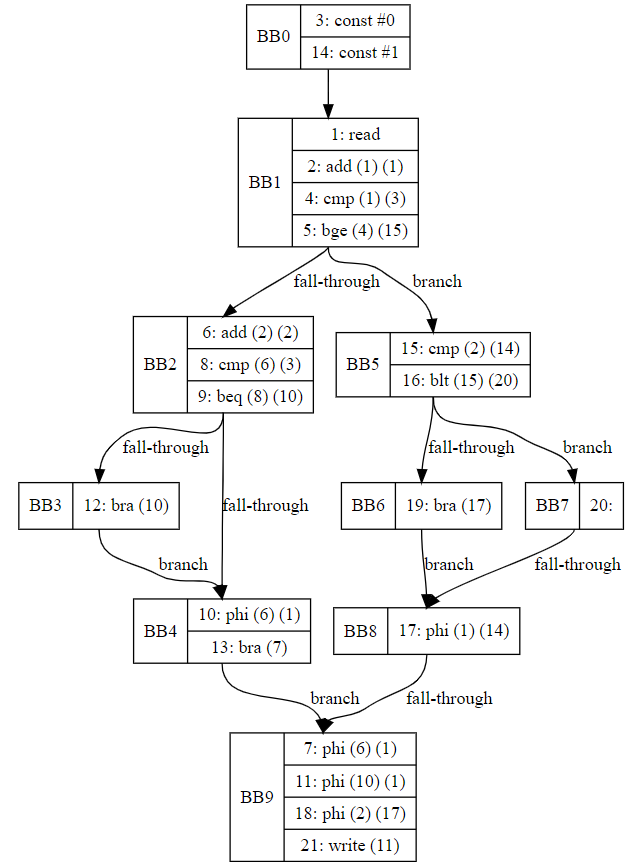
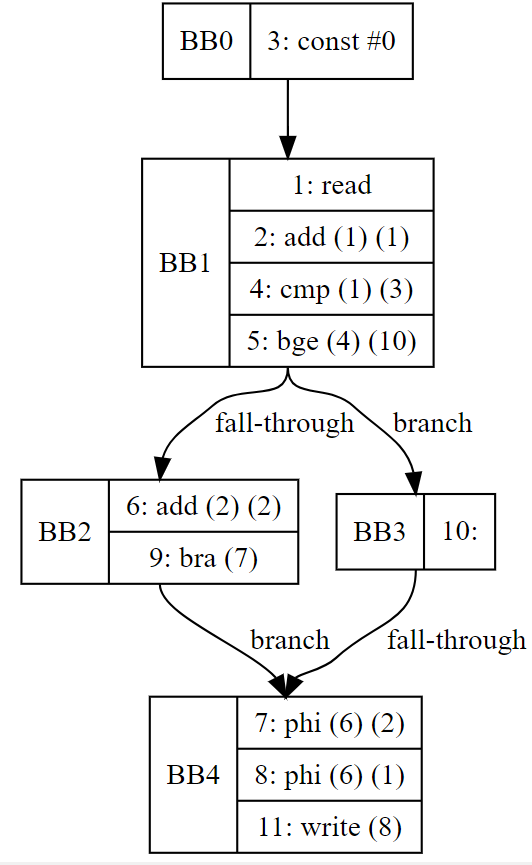
Clustering 86(R3) 1(R1) 72(R2)

## Half-Time Project Report

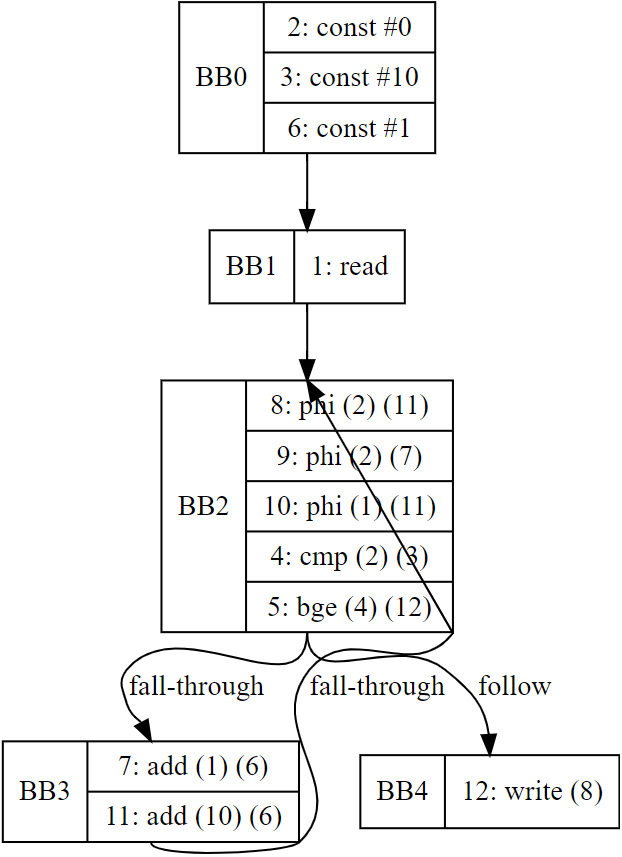
I’ve implemented the basic skeleton of the EBNF for the main project, including regular language and some context language. Most parts of the regular language are completed except for the array designator.

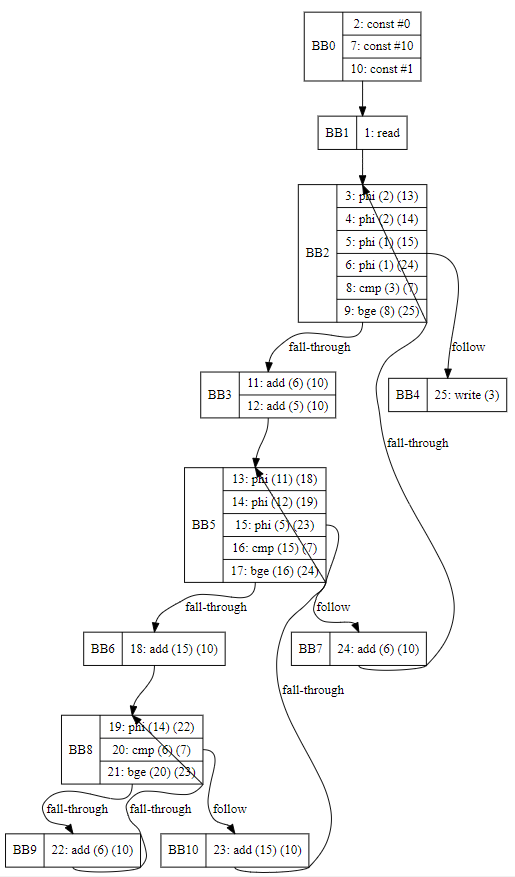
Among the context language, I completed assignment, ifStatement, typeDecl, and varDecl. Because I am doing the project individually and don’t need to add the user-defined function, I commented out the funcDecl, formalParam, funcBody, and returnStatement for now. The funcCall is kept because we have the predefined functions.

So far, the main focus of the main project was the if-else statements. I’ve tested my program with the provided example (left) and with my example (right) and both generate the right diagram (as shown below). The nested if-else example can be found in my source code.

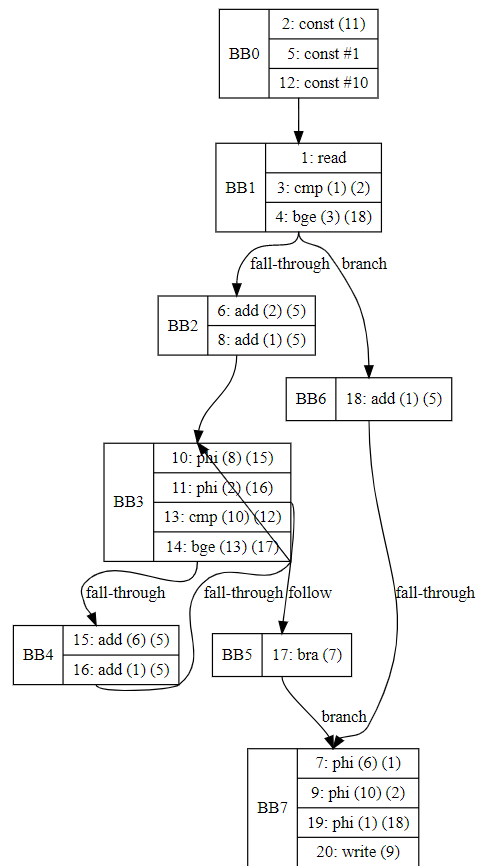


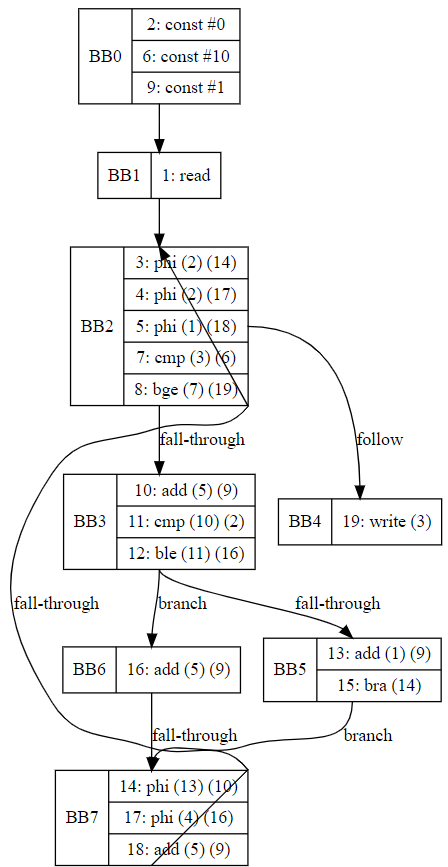
In sum, I’ve finished project step 1 and 2 except for the while loop. My goal after this will be adding the array and while loop. I will also be testing the program with more complex if-else examples to ensure it’s working correctly.

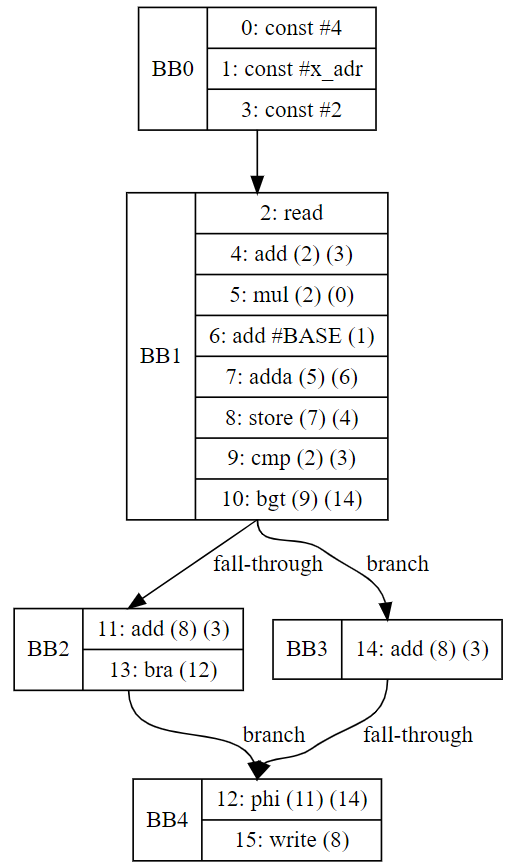
While 

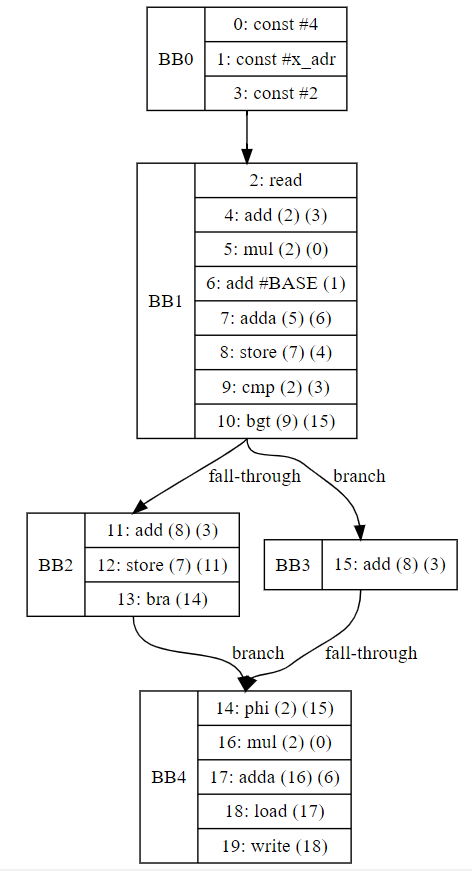


3 Nested While:

While in if

if in while 

Simple array with if

Array assignment in if-else

Array in while with index changed (left) and without index changed (right)

