```
int main(){
    return 0;
}
```

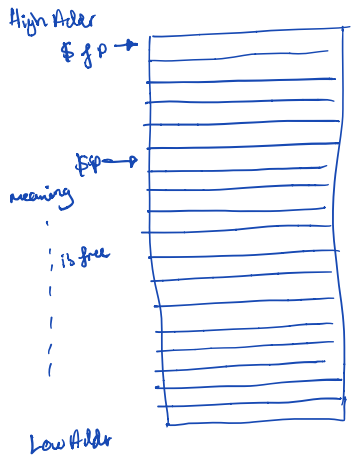
⇓

```
addiu   $sp, $sp, -8
sw      $fp, 4($sp)
move    $fp, $sp

move    $2, $0

move    $sp, $fp
lw      $fp, 4($sp)
addiu   $sp, $sp, 8

j       $31
nop
```




| register | assem name | comment |
|---|---|---|
| r0 | $zero | always 0 |
| r1 | $at | reserved for assembler |
| r2-r3 | $v0 - $v1 | stores results |
| r4-r7 | $a0 - $a3 | stores arguments |
| r8-r15 | $t0 - $t7 | temporaries, not saved |
| r16-r23 | $s0 - $s7 | contents save for later use |
| r24-r25 | $t8 - $t9 | more temps, not saved |
| r26-r27 | $k0 - $k1 | reserve for OS |
| r28 | $gp | global pointer |
| r29 | $sp | stack pointer |
| r30 | $fp | frame pointer |
| r31 | $ra | return address |

MIPS Stack Pointer and Frame pointer

Stack pointer = points to the top of the stack as you allocate space, the stack pointer ($sp) moves to point to the free memory

Frame pointer = points to the start of the stack frame and does not move for the duration of the subroutine call. Points to base of stack frame, + parameters are passed in to the subroutine remain at a constant spot

# WHAT MAKES IT HARD

→ STRUCTURE HARD TO CHANGE

→ YOU THINK IT'S HARDER THAN ACTUALLY IS

SETUP

```
addiu   $sp, $sp, -8
sw      $fp, 4($sp)
move    $fp, $sp
```

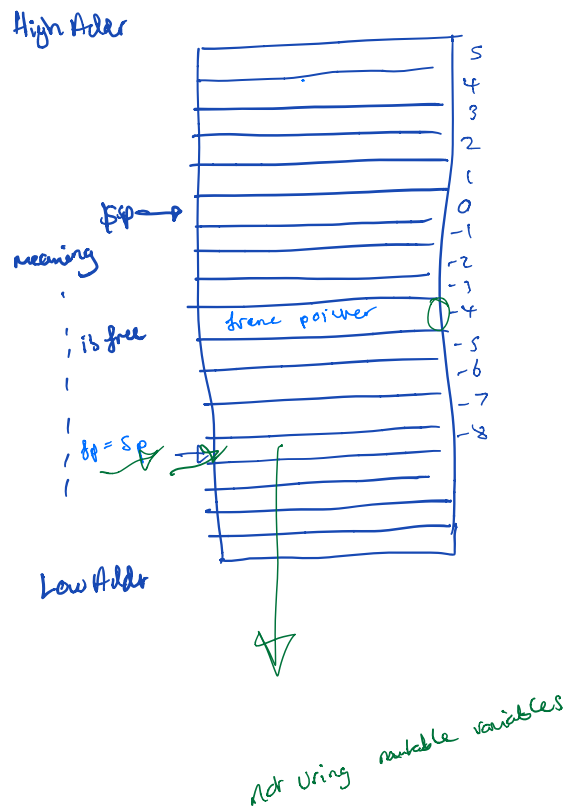
OPERATION

```
move    $2, $0
```

RESTORE

```
move    $sp, $fp
lw      $fp, 4($sp)
addiu   $sp, $sp, 8
```

```
j       $31
nop
```




LEXER → PARSER → C_translate, P_translate, C_compile

C_compile

program. — } keeps control of stack frame pointer
program. — } etc
program. —
instruction (- - - -) → keep control of output creating instructions

Node
- c_translate( dst )
- p_translate ( dst ) // incl environment soon
- c_compile ( context , dst )

context (program)

offset - how does this work
frame pointer
stack pointer
map of variables

need to include:
- .globl $(function label)
- .align 2
- .set nomips16
- .set nomicromips

Bugs

Structure

compiler - - - -

f:

all instructions after label have indentation two spaces

First five test cases

```
int f(){
    return 0;
}
```

```
addiu   $sp, $sp, -8
sw      $fp, 4($sp)
move    $sp, $sp
li      $2, $0
move    $sp, $sp
lw      $fp, 4($sp)
addiu   $sp, $sp, 8
j       $31
nop
```

Basically want to

1 * 5 + 2 * 3

Store result

((1+2) - (3*4)+2) + 3) + 4)

---

```
int f(){
    return 1+2;
}
```

```
addiu   $sp, $sp, -8
sw      $fp, 4($sp)
move    $sp, $sp
li      $2, $3
move    $sp, $sp
lw      $fp, 4($sp)
addiu   $sp, $sp, 8
j       $31
nop
```

---

① First thing to do will be parentheses the equation to see how I parse!
2nd work out registers, stack use

② Try passing many parameters, 1, 2, 3, 4, 5, ..... to see how stack behaves initially

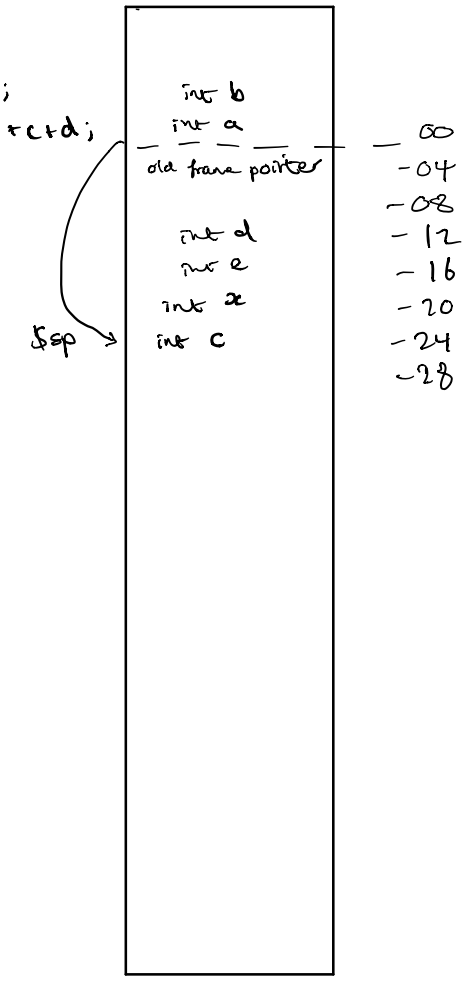
Argument passing, stack test

| Arg no. | info | |
|---|---|---|
| 0 | initial -8 | |
| 1 | initial -8 | register 4 store $fp + 8 |
| 2 | initial -8 | ↑ repeat and register 5 store $fp + 12 |
| 3 | initial -8 | ↑ repeat and register 6 store $fp + 16 |
| 4 | initial -8 | ↑ repeat and register 7 store $fp + 20 |
| 5 | initial -8 | idk |

Parameter numbers

| Param no. | | |
|---|---|---|
| 0 | -8 | |
| 1 | -16 | |

```
int f(int a, int b){
    int c = 1;
    int x = 100;
    int e = 5;
    int d = 1;
    return a+b+c+d;
}
```

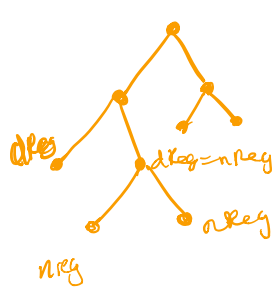



Registers

| | | |
|---|---|---|
| 0 | zero | 0 |
| 1 | at | |
| 2 | v0 | return val |
| 3 | v1 | expr eval + return val |
| 4 | a0 | passing arguments (up) |
| 5 | a1 | |
| 6 | a2 | |
| 7 | a3 | |
| 8 | t0 | temp registers for expr eval |
| 9 | t1 | |
| 10 | t2 | |
| 11 | t3 | |
| 12 | t4 | |
| 13 | t5 | |
| 14 | t6 | |
| 15 | t7 | |
| 16 | s0 | saved registers values preserved across f calls |
| 17 | s1 | |
| 18 | s2 | |
| 19 | s3 | |
| 20 | s4 | |
| 21 | s5 | |
| 22 | s6 | |
| 23 | s7 | |
| 24 | t8 | more temp reg |
| 25 | t9 | |
| 26 | | |
| 27 | | |
| 28 | gp | global pointer |
| 29 | sp | stack pointer |
| 30 | fp | frame pointer |
| 31 | ra | return address |

START

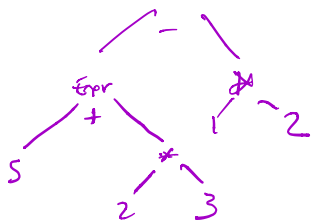
| dReg | nReg | Lcount |
|---|---|---|
| 2 | 2 | 0 |




int dReg = nReg : dReg=2

nReg inc

Expr : (5 + (2 * 3)) - (1 * 2);




Way 1: <u>Either expression</u>

# Addition Expression

reg = get destination register

prog-context

return reg

return reg + 1

context.setDestReg(dest_left)

go to left node → print_mips(prog)

fp = increment frame pointer

sw dest left, ----

dest_right, ---- print mips

lw dest_left

addu destreg, return reg, return reg+1

Function Definition

Noheptor

$$ Declarator — dec NT

$$ Direct Declarator

Empty Declarator / Paramlist Declarator — Direct_Declarator T_LBRACK T_RBRACK

Variable Declator — Direct Declarator T_IDENTIFIER →

"

→ String

I could have

Declarator

Function Declarator with
virtual void get label

Next

- clean up classes
- sort out python
- sort out expressions!

DECLARATIONS

EXPRESSIONS

ARITHMETIC
binary
EXPRESSIONS

POST/Prefix

STATEMENTS

Compound Statement

Declaration

CONSTANT

Declarations-

Statements

dec →

stmt →

stmt →

dec →

```
struct environ {
    int scope_count;
}

increment_scope_count(const environ & cur) {
    environ x = cur
    x.scope_count += 2;
    return x;
}
```

if( ) {

statement{

→

}

nestor of

name → offsers

using stack pointer

e.g. x → 0

y → 4

Maybe worth addining Arithmetic expression to classes

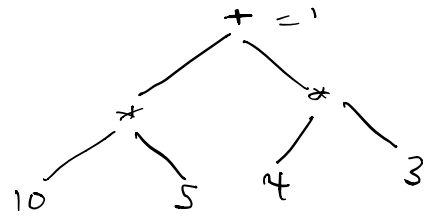
Arithmetic Expressions

lhs rhs

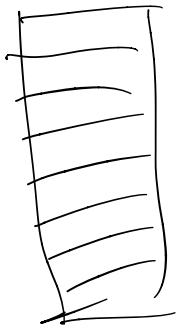
get_type?

return 10 * 5 + 4 * 3

register_avail = 0




dest reg as return is 2



dest reg = 1 if
avail reg =

arithmetic_expression get_OType()
print_cmips( dst , context )

reg

Steps to do for expressions:

Context passing by value rather than reference

add OType

PLAN:

return Expression

if OType = Arithmetic expression

Test case

int f() {

return 1*2 + 3*4;

}

RETURN Expression T_SEMICOLON

don't want const context/prog

context & prog

Return Expr Statement

destReg = 2 availReg = 8

Expr → print

availReg = 8

b return label

nop

print → Expression

+

2 3

left takes dest Reg

right take avail Reg

d

e

print

((5*3) + (4*5)) - 3) + (4*3))

ldeg rdeg

d = 2

e = 3

addu d, d, e

ldeg rdeg incr

ld = d

d

ld = 15

ld.avail rd.avail

d.incr

Add

ld = d;

Leval

ld.incr

rd.incr

r.eval

li d, 5

li d, 3

mult d, d, d.incr

LEFT

lhs → print

RIGHT

COMBINE

Function add

dth = d

lhs → print_mips

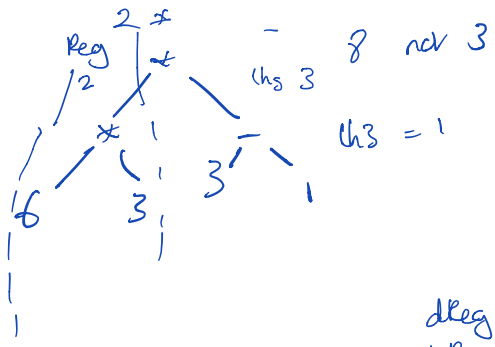
dth.incr

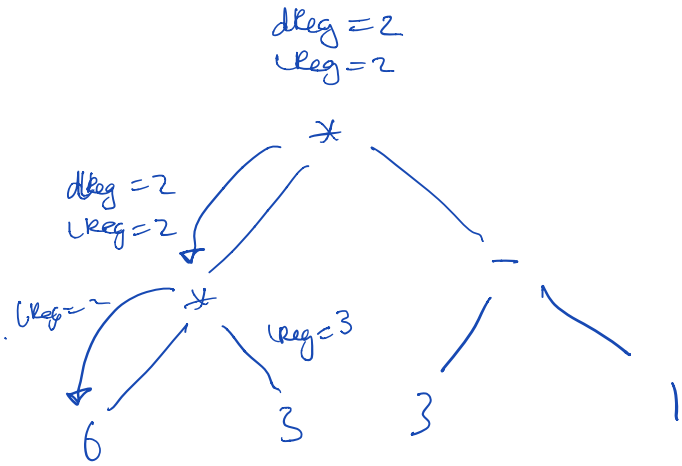
drh.incr

rhs → print_mips
addu d, dlh, drh

## Expressions

6*3*(3-1)




## Expressions

*
/
%
+
-
<<
>>
>
<
<=
>=
==
!=
&
^

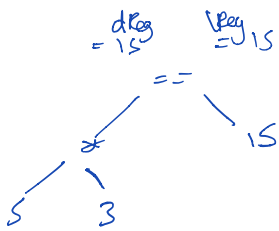
<=

3 <= 2 slt
1 10 xori dReg, 1
1

0 0 0
0 1 1
1 0 1
1 1 0

2 <= 3

5*3 == 15




10 << 2

0 0 1 0
1 0 1 0
1 0 1 0 0 0

32 8

40 10
8 2

1 0 1 0
0 0 1 0 2
0 1 0 1 7

1+4=5

|
( )

```
beq   dReg, cReg, label
nop
move  dReg, 0
b     end
label:
li    dReg, 1
end
```

12/03/18

① Get all arithmetic instructions sorted
② Building tests as I go along

- lexer tokens
- parser, classes
- print_mips
- tests

③ Local variables stack
④ Mix with expression

⑤ Maths CW

$a > b$

$b < a$

SLT rd, rs, rt

rd ← (rs < rt)

5

$(((19 < 11) + (2 * 3)) - 1) + (3 == 1)$

0 6 -1 0

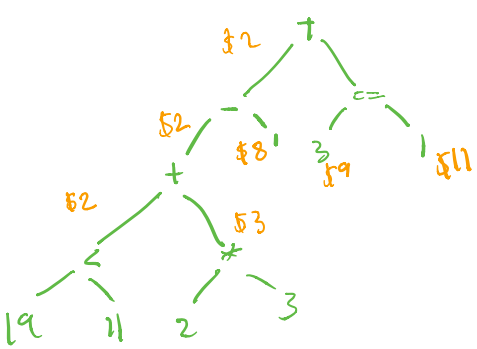
2=0

6

0 + 6 - 1 + 0

= 5

addu $2, $2, $3 → $9



LOCAL INTEGERS

```
int f(){
    int a = 5;

    return a;
}
```

Variable declaration

- type
- name
- =
- expression

scope

variables < name, value >

assign_var()  variables < name, value, position >

free_var()

```
int f(){
    int a;
    int b;
    int c;
    int d;

    return 5;
}
```

Compound Statement

decl list

| | | | |
|---|---|---|---|
| int | a | = | 5 |
| int | b | = | exp |

decl list

- dec
- decl list
  - dec
  - decl
    - dec
    - dec

decl list
stat list

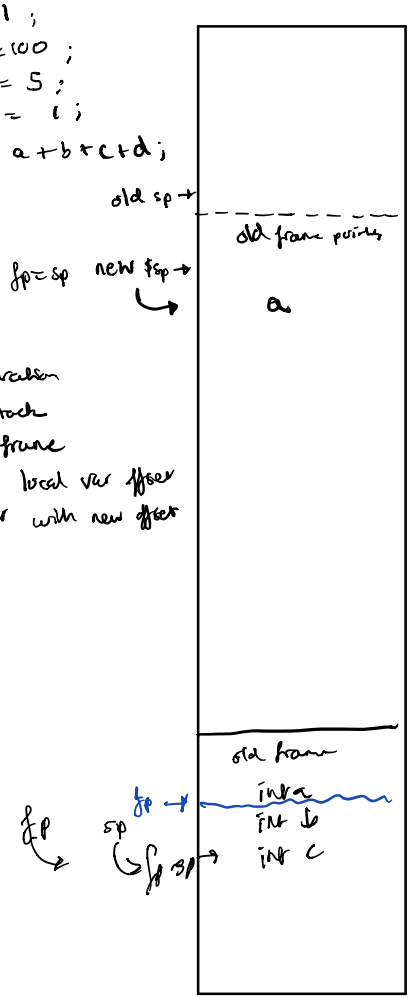
fp
sp

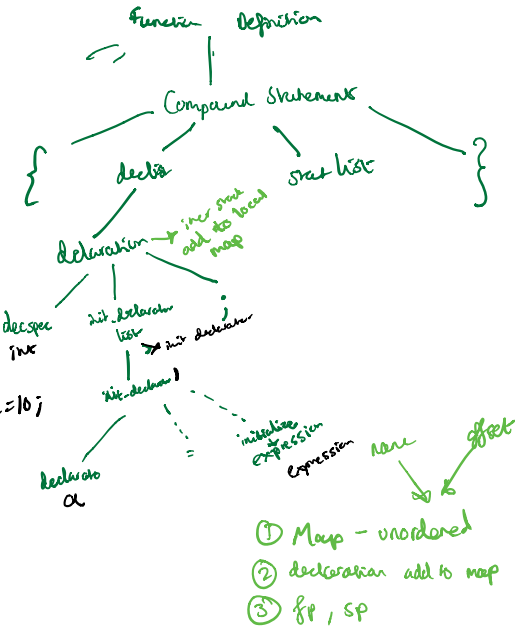
int a = 2;
int a 5;

34.36

return a; 5 + 3 + a

```
int f(int a, int b){
    int c = 1;
    int x = 100;
    int e = 5;
    int d = 1;
    return a+b+c+d;
}
```



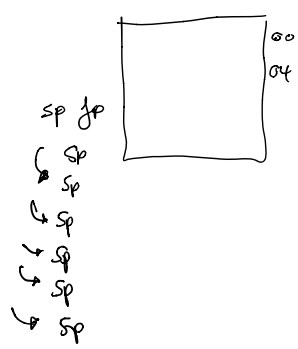




declaration
bump stack
move frame
update local var offset
add var with new offset

int a = 5

(a → 04)

create offset




FRAME SIZE - OFFSET

```
addiu $sp, $sp, -8
sw    $fp, 4($sp)
move  $fp, $sp
```

```
declaration {
    if when already a frame
        replace value
    else
        increment frame size   offset = frame size
        push onto map
```

# declaration

STACK SIZE

as same size as frame

TODAY

11.30 → 2pm
→ declist sorted → tests, → list, → skar
→ if, for, while ?

MATHS CW
→ Finish code look through
→ Report part 1
→ Check others part

$6 + 12 + 2 = 20$

Tests

```
int f() {
    int a = 5
    if (a > 5) {
        a = a + 1;
    }
    else {
        a = a - 1;
    }
    return a;
}
```

a = a + 1

mips-linux-gnu-gcc -march=mips1 -mfp32 -static -c -S main1.c -o main

Assignment Expression

unary expression — assign op — assign expression

a — = — a + 1

get name

assignReg() $3
expression result in $3

mov $2, $3
sw $.

a++ 1

a += 1

a[2] = 2

a = a + 1

$2 $3

int a;

"a" is 0.

a = a+1;

clean_var

Unary_Expression

↓

Post_fix Expression

↓

Primary_Expression

↓

Identifier

Assign Expression needs get name

create inter

OR

add get_name in all and write not implemented

class Assign

print

Statement of a = get

a → get_name ( )

get_name

Expression

getvname

Assign

Expression

Assignment Exp

Conditional

Assign Expr

Unary Expr
postfix expression

identifier | [ ' ' ]
| ( )

incop
++
--
& unary
size of

Operator
=
+=
-=
*=
/=
%=
&=
^=
|=

Assign_Expression
anything ! ! ! !

Assign

Unary Expr

Expression Var

prefix → identifier
postfix → identifier

Variable declarator

Create Identifier class

let inherit from var unary expr
expr var

AssignExp

Unary Expr

Node — Expr

class Identify : public Expression {
private:
get_ID ()

Expression

Identify

primary_Expr

postfix

}

Which is
UnaryOp Cast_Expr

STEPS

① Assign class to get AssignExpr working

a++ a &= ...

② If else statement

if else
if else

Identifier

Lone Post

Identifier

class A

int get_ID  const=0

B : public A

getID

Expression Statement

a=a+1;

Input

int f() {
int a=5;
~~a=a+1;~~
return a;
}

③ while

④ For

⑤ Argument passing  Parameter

⑥ Global var

⑦ Function Calling

⑧ Recursion

⑨ a++ — b++ . . .

⑩ Pointers

⑪ Arrays

⑫ Strings

⑬ Floating-point

⑭ Structs

If else statement

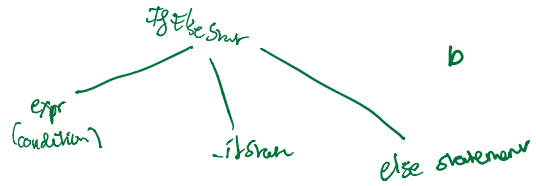
beq ---- elselab rs, rt, offset

nop

b end

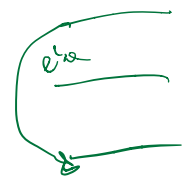
beq

if




anything except 0 is true

bne 0 $t5 true




While statement

(condition is true)

Condition stmt check

label

Stmt ----

condition check

b $L2

nop

$L3:

$L2:

load condition

nop

bgtz $dest, $L3

nop

# FOR STATEMENTS

for ( express_statement express_statement ) statement

for ( expression_stat expr_stat express ) statement

for ( declaration expr_stat ) statement

for ( declaration expr_stat expr ) statement

for(int i=0; i<10, i++)

for(

expression

if f()){

return 0;

// TO-DO - +

return

i++;

i--;

i ++

postfix expression

operator

i=i+1

int f() {

int a = 2;

a++;

return a;

}

==3

int a=1

for ( a=0 ; a<10 ; a++ ) {

expr stat (assing exp)

expr stat (expression)

expression

}

Function Definition

def f : dec

convenient indented

The declaration

int a = 5;

return a; return b;

return c;

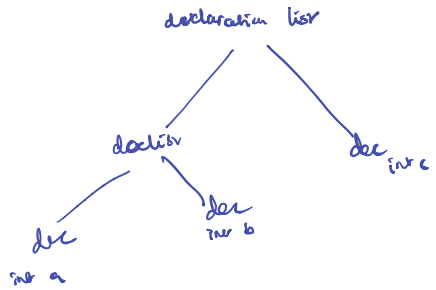
function definition

def .....:

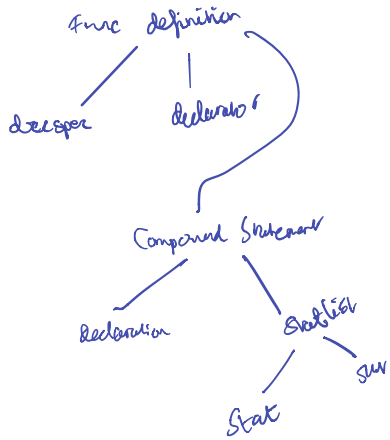
compound statement

dec

return

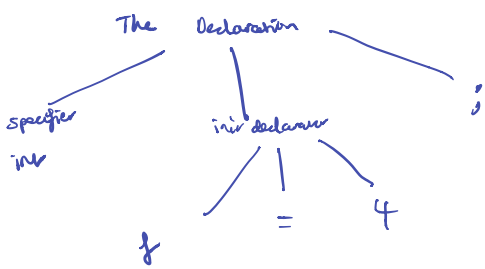



```
int main() {
    int z;
    z=7;
    return z;
}
```




dec

init declarator




```
z=7;
return z;
```

```
int z;
```

```
{
    if(0)
      if(5)
        return 10
      else
        return 13;
    return 11;
```

NEXT → Parameter passing f(int a, int b, int c, int d)

→ Global variables

→ Scopes

within function
f(11, 12, 13)
postfix (arguments, exp, to)

Parameter passing

Function Definition

int f(int a, int b, int c, int d) { }

decspec

Declarator ---- Empty Declarator
print_mips

Param List Declarator
Param List Declarator

Param Declaration

Param List

Param Declaration

Param List

Param Declaration

Param Declaration

dec

decspec

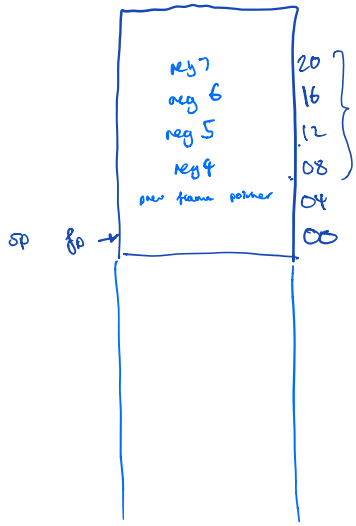
compound Statement

func definition add param number

public

public Param Pass

Context
int parampass = 0
int parampos 1 × 4 + 4




① Enter function definition
② add parameters to stack
add parameters to map

frame size + 4 + (param - 1)

Global variables

int a = -5

int a = 5

Stack Frame

space for
1. local variables + temporaries
2. saved general registers
3. saved floating point

Negative numbers

unary operator cast expression

-
+
&
*
~
!

- does →

Unary Exp :
Postfix
-
+
*
cast-expression

unary minus expr
unary plus expr
unary times exp :
⋮

hex input etc

```
.globl a
.align 2
.type a, @ object
.size a, 4
a:
  .word 5
```

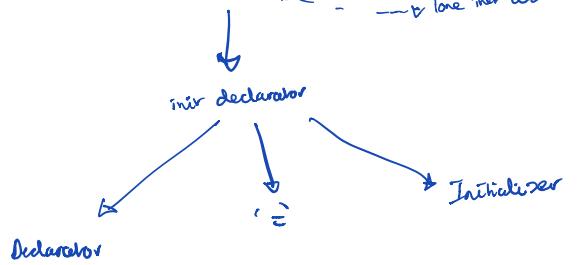
• Stuff

is global or not ⇒ scope number

① program scope counter

② struct {
var name :
scope :
unique name :
}

The declaration




→ lone init declarator

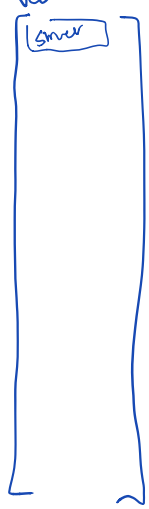
Context
• scope

The Declaration ⇒ ① Check what scope
② If global
prior all info
③ go in and evaluate
⇓ from then on if (scope = 0)
do not use register

variable name
scope

Var

Vector




Variable map

| scope num | var name |
|---|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |
| 7 | h |
| 8 | i |
| 9 | j |

check var character & smthg

Context
scopes

after every exit delete
current scope var

ID —

CHANGE localMap → VarMap

std::unordered_map<std::string, int> VarMap

pair

key → (string, int)

func(a,b)
return c

std::  (int, uno...(  ))

finding if there

$a \to b \to c$

$\equiv a \to (b \to c)$

type T
include typinfo.tee
of Int

check Var → does it exist

int a;

int a=2

Types of Declaration

The Declaration

decspec

init-declarator list

InitDeclarator

;

InitDeclarator

Assignment expression

Unary Expression

=

expression

Expression Variable

InitDeclarator case

int a = 2;

① Check if in current scope
① if yes change
② else add

① if declared before but not global return offset

② if declared in global global stuff

③ Else add

int → < String, int >

d[ ]

m ∈ map<'a', 'b'>

m[a] ∈ b

m[a][b]

m[scopenum][name] = offset

map<'a,'b>

.at(a) → 'b

[a] → 'b

.b()

m.erase(~~stop_iter~~)

~~m.e~~

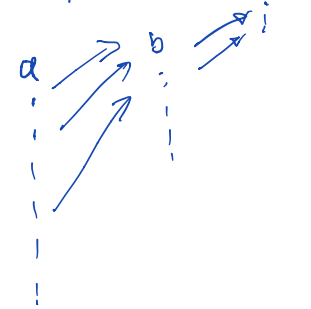
m.erase(m.begin

for map

find

x

<a, <b,c>>




find

it->second is

x[name] -> c

-> var = number
var expression

MONDAY

1. Global variables → evaluate → add all crap
2. Function calling
3. Delete Scope

Global variables

- .globl a
- .data
- .align 2
- .type a, @ object
- .size a, 4

a :

- .word 5

if no value = 0

program.setnReg()

programiser

if not declared before

AssignEqExpr

Variable

int x = 5

func ( )

int x = 2

if global variable need address

Function Calling Mips

int f( )

return f(1,2,3,4,5,6);

Non-Leaf function

at least four words must be allocated

f(1,1) + f(0,0)

function must put arguements on bottom of stack

int f() {
return 10;
}

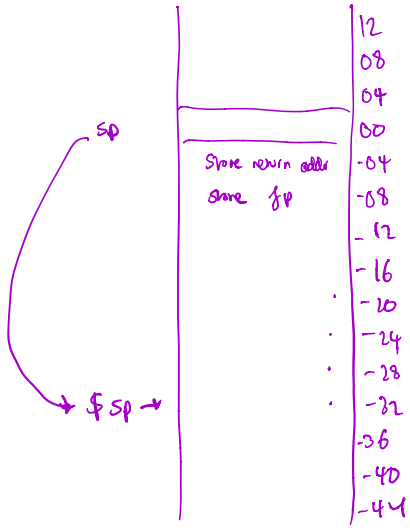
f() → LonePostfixExpression

f(.....) → PostfixArguExpression

int main() {
return f();
}

LonePostfix

```
int main() {
    return f();
}
```



40mins

→ 2.30

destReg = 3

don't use reg 2 for expressions cuz it sign at level due to functions

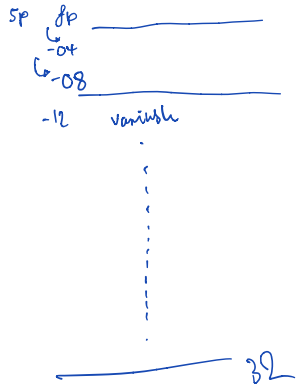
f() + f() + f() ...

reg 2

Stack allowance

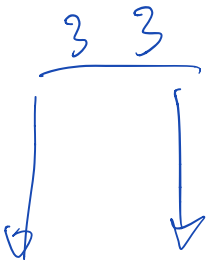
when function call add everything w stack then reload on completion

Currency Stack




Function use

allocate frame of

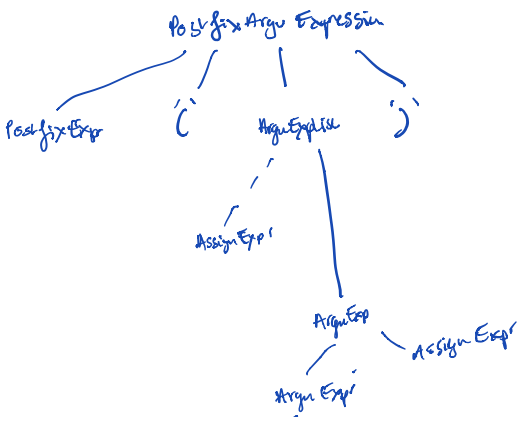



+

+ nested calls
many, many, many ...

Steps

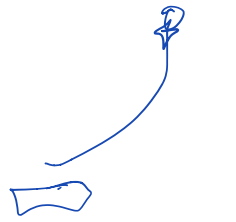
- change stack places
- write mips for function calls
- write with parameters passing
- write with expressions
- save registers to stack and back



Change AssignExprList to have NULL option

AssignExpr — Assign

f( 3+4 , 2+1 )

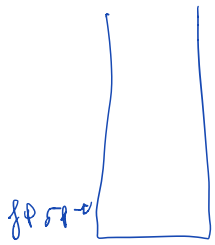


## Function calls

10:40 → Comms

① Save registers to stack and back again
find useful registers
gone all do f()
restore all registers

② Improve Testing
③ Finish best deliverable section
④ Arrays, Pointers .....




| Registers | | |
|---|---|---|
| 0 | zero | 0 |
| 1 | at | |
| 2 | v0 | return val |
| 3 | v1 | expr eval return val |
| 4 | a0 | passing arguments (up) |
| 5 | a1 | |
| 6 | a2 | |
| 7 | a3 | |
| 8 | t0 | temp registers for expr eval |
| 9 | t1 | |
| 10 | t2 | |
| 11 | t3 | |
| 12 | t4 | |
| 13 | t5 | |
| 14 | t6 | |
| 15 | t7 | |
| 16 | s0 | saved registers values preserved across f calls |
| 17 | s1 | |
| 18 | s2 | |
| 19 | s3 | |
| 20 | s4 | |
| 21 | s5 | |
| 22 | s6 | |
| 23 | s7 | |
| 24 | t8 | more temp reg |
| 25 | t9 | |
| 26 | | |
| 27 | | |
| 28 | gp | global pointer |
| 29 | sp | stack pointer |
| 30 | fp | frame pointer |
| 31 | ra | return address |

f09 → expected 35 got 52

13 + 2 + 13 + 7 = 20+15 = 35

52 →

13×4 = 52

if (a<b)
here is the problem

f10.c expect 1, get 139

- need to leave 4 words for function params

franklin

if

if

$19 * 4 = 76$

$+16$

$= 92$

$23 * 4 = 80$

$3 + 2 + 13$

$23$

Postfix argumentexpression

$f(2, 3, 4)$

AssignExprList

## TODO

C

- pre, post increment
- —, hex etc
- short circuit
- +=, --a, ----
- function call in if(___)
- right stl, srn ....
- test deliverable section
- go through to check parser + lexer capabilities
- paramlist

Python

- Ensure while is done
- empty → pass
- parameter list

Arrays

- Pointers
- Floats

Standard Additions

- UnaryOpExpr —

Unary Expression

un_op cast_expression

this is under assign_exp → unary assign op = assign expr

AssignExpr

int a = $\boxed{-5}$

cast expr

unary expr

unary op

cast expression

Change Unary Expression to one class not unary_op

- converting hex input to value etc

- a - -

- fixing right shift (keep >> str etc)
- conditional expression
- logical_or_expr '?' expr ':' conditional

AssignExp

*= /= %= += -=

<<= >>= &= ^= |=

→ add keys for | and &

Expression : assign exp (for Python as well)

| expr , assign_expr

int a, b, c ; ....

may be identifier list also

Parameter_List → ParamList()

Identifier List not in parser

Case for CompoundStatement(NULL, NULL)

(Python NULL)

iteration statement & empty conditions

short - circuiting

Medium

- declarator : pointer direct_declarator
- direct_declarator : [ ]
- or designator list?
- passing f( a, b, c) instead of f(1, 2, 3)
- Switch ( expression ) statement

for a(5)

when returning effect

return never effect

eg a0 → a1 → a2 →

0 1 0 0

2D arrays

Weird Additions

- func_def : dec_spec declarator compound statement

int a { }

- ... ellipsis
- param_dec : dec_spec abstract dec
- casting ('type_name') cast expr
- constant expression
- Initializer_List

Structs unions

- labelled statement
- jump statement
- Skipped all of block_item
- expr_statement : ;

[

TO-DO

- post vs pre incrementing
- global currency - hard
- passing f(a, b, c) ... in functions
- hex
- pointers
- ensure list works for assignees
- iteration statement empty conditions
- short circuiting

~~Test deliverable~~ → int f();

- Empty iteration
- hex
- pointers

16 bit currently

*a

a + 7

Parser

Tokenizer

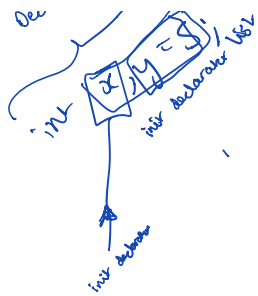
Tokenizer

# Arrays

a [10]

store at offset

increase the stack size by 10*4




1. support lexer
2. support parser
3. in functions ensure correct offset stack
4. in functions ensure correct element accessed

Array declarator

don't use identifier to store

use get_name ( )

Declaration

don't really worry about this as support for integers only

Declaration-specifiers

Init-Declarator-List

;

The Declaration

Init-Declarator

Init Declarator have Init Declarator

Declarator

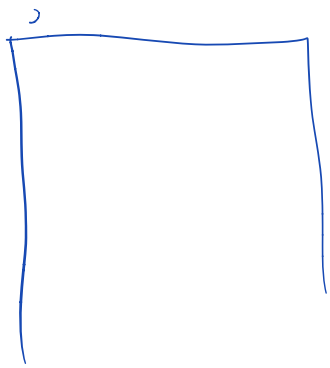
'='

Initializer

return exp

return postfix

for(int i=0 ; i<10 ; )



a[0][0]
a[0][1]
a[1][0]