Overview

Synthesis stage = Code Generation + Optimization

Goal: generate assembly code for the assembler

Input	type-annotated parse tree
Output	.asm
Next Stage	Assembler

Infinitely many MIPS programs for a single WLP4 program

Efficient Compiler (2 ways to interpret)	
in the time it takes to compile	Generates efficient code
What we use	Better Method

Mapping Variables to Locations

mips.twoints convention for wain

- $1, $2 = parameters
- $3 = return

```
int wain(int a, int b){
  return a;
}
```

The structure of the parse tree alone is not enough to determine what output to produce.

Extend the symbol table

With a **location** entry for each variable

- locations on the **stack**
- **location:** offset from **Frame Pointer**

With a location entry for each variable:

- traverse the parse tree
- if we encounter a variable, refer to the location entry

Frame Pointer (FP)

FP = $29

- **location** of the **first value pushed on the stack by the caller**
- does not change within a procedure
- Each procedure has its own FP
- All variables are referenced with respect to FP
- Changes to the stack pointer have no effect on the offsets.

```
int wain(int a, int b)
{
    int c = 0;
    return a;
}
```

Symbol	Type	Offset from $29
a	int	0
b	int	-4
c	int	-8

```
lis $4
.word 4
```
} convention to store 4 in $4

Prologue

$29 = $30 − 4

sw $1 → 0($29)
$30 − 4

sw $2 → -4($29)
$30 − 4

int wain(int a, int b)

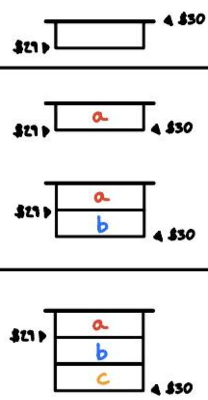
Code

sw $0, -8($29)

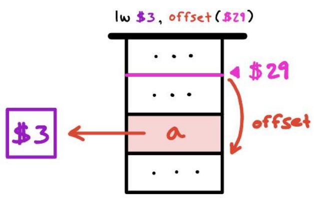
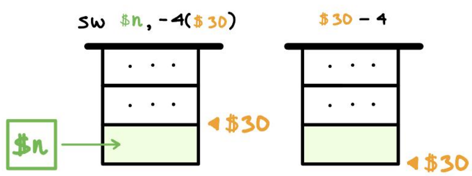
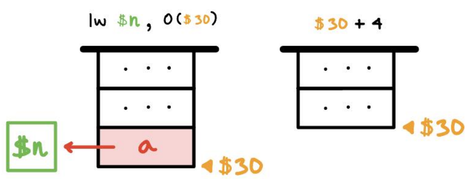
int c = 0;

lw $3 ← 0($29)

return a;

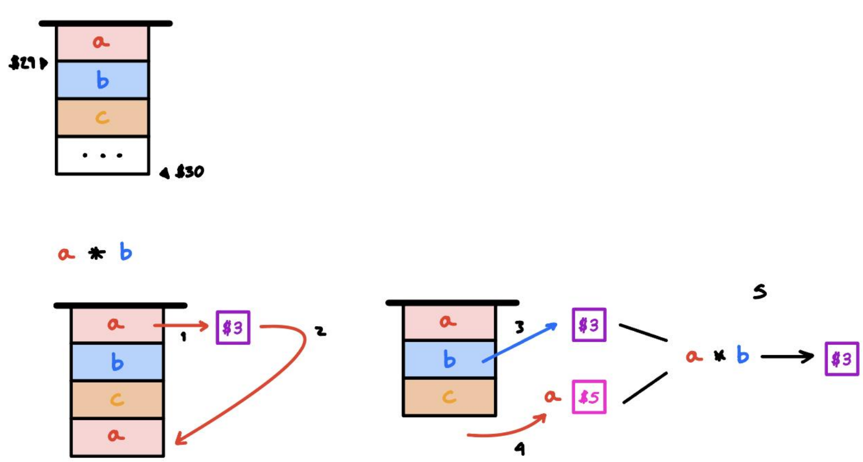


Expressions

Shorthands	
code(a)	
push($n)	
pop($n)	

Expressions, Universal Code Gen Scheme

uses one temporary register ($5) to compute any arbitrarily complicated expression

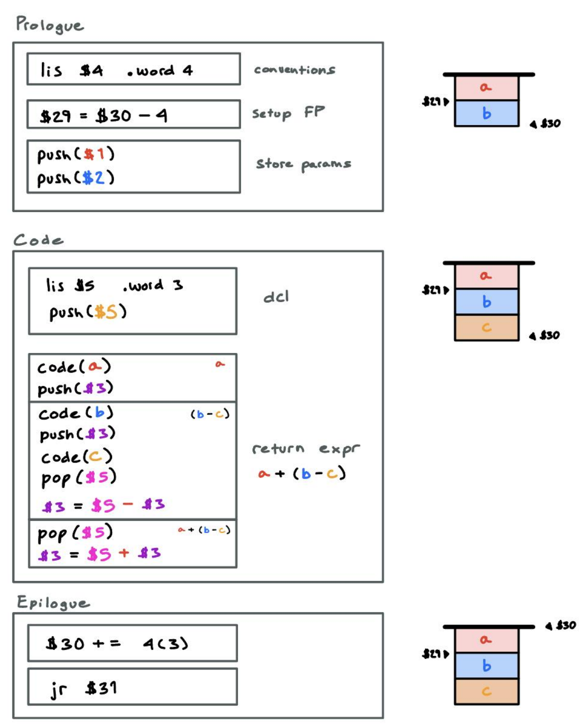
1	**code** (a)	
2	**push** ($3)	
3	**code** (b)	
4	**pop** ($5)	
5	*$3* **=** ***F(****$3***,** *$5****)***	

Step 3

1. **‘b’ is a single value:** code(b) = $3 = b
2. **‘b’ is an expression:** repeat steps 1-5

Example

```
int wain(int a, int b)
{
    int c = 3;
    return a + (b-c)
}
```



Shorthand For Rules

Same for: addition, multiplication, subtraction & modulo

expr	→	expr_2 **PLUS term**
code(expr_1)	=	**code** (expr) **push** ($3) **code** (*term*) **pop** ($5) **$3 =** $5 + $3

Easy Ones

S	→	⊢ **procedure** ⊣
code(S)	=	**code** (procedure)

expr	→	**term**
code(expr)	=	**code** (term)

factor	→	**(expr)**
code(factor)	=	**code**(expr)

factor	→	**(expr)**
code(factor)	=	**code**(expr)

Variable Assignment

statement	→	**lvalue BECOMES** expr **SEMI**
code(factor)	=	**code**(expr) sw $3, offset($29)

ie: x = 5 + (2 * 3)

Print Statement

Rule:

statement	→	**PRINTLN** LPAREN expr RPAREN SEMI

ie: print(2+3);

Note: expr is an 'int' (from CSA)

Overview:

- The compiler imports a print-procedure
- assumes that this procedure is available in the runtime environment
- The compiler then calls this imported procedure every time a println statement is used
- The pre-compiled code in the runtime environment is **linked** with the output that the compiler produces.
- This linking process produces the standalone executable

Runtime Environment

Runtime Environment:

- contains procedures, libraries, variables, that we can use.
- Assist programs during execution

Object Files

The pre-compiled code in the runtime needs to hold info beyond just the code for 'print'

It must also "announce" some things:

1. what the code provides
2. what it expects.

This information is not MIPS code

Similarly, the compiler generated code must "announce" that:

1. print' was assumed to be available.

Object File:

1. Compiled code
2. Additional info
    a. What the code **expects**
        i. **ie:** announcement that this code expects to be linked to a *print* procedure
    b. What the code **provides**

MERL

- MIPS object files
  - MIPS machine code,
  - info for *linker* & *loader*

During code-gen, if the compiler needs to use *print*:

it will generate an **assembler directive** to import print at the top of file:

```
.import print
```

The result from **code-gen** is an assembly file + import statements

This means that we must now use an assembler that understands imports ⇒ **Linking Assembler**

Linking Assembler

cs241.linkasm

- **Assembler** that understands **imports**
- produces object files (rather than just MIPS machine code)

```
cs241.linkasm < output.asm > output.merl
```

Linking

Take multiple of object files

- links them
- produces a new object file which contains the combined machine code and announcements

```
cs241.linker output.merl print.merl > linked.merl
```

To produce the pure MIPS machine code, we must strip out the MERL metadata from the object file

Produce Machine Code

cs241.merl

- strip out the MERL metadata from the object file

```
cs241.merl 0 < linked.merl > final.mips
```

Result

We can generate code for the WLP4 *println* statement by knowing what the print-procedure we will be linking with expects:

println:

- **expects** input in register 1

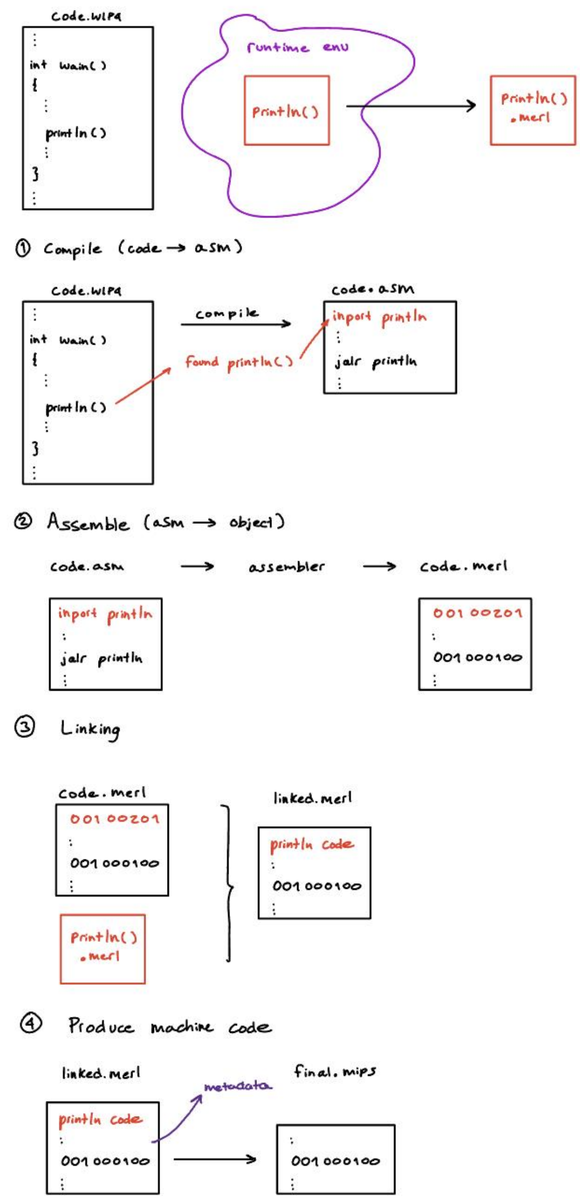
Just like any other procedure, we will call print using the jalr instruction which overwrites $31

statement	→	**PRINTLN LPAREN expr RPAREN SEMI**	
code(statement)	=	push ($1)	preserve $1
		code (expr) $1 = $3	expr = input to println
		push ($31) lis $5 .word *print* jalr $5 pop ($31)	call println()
		pop ($1)	restore $1

Summary

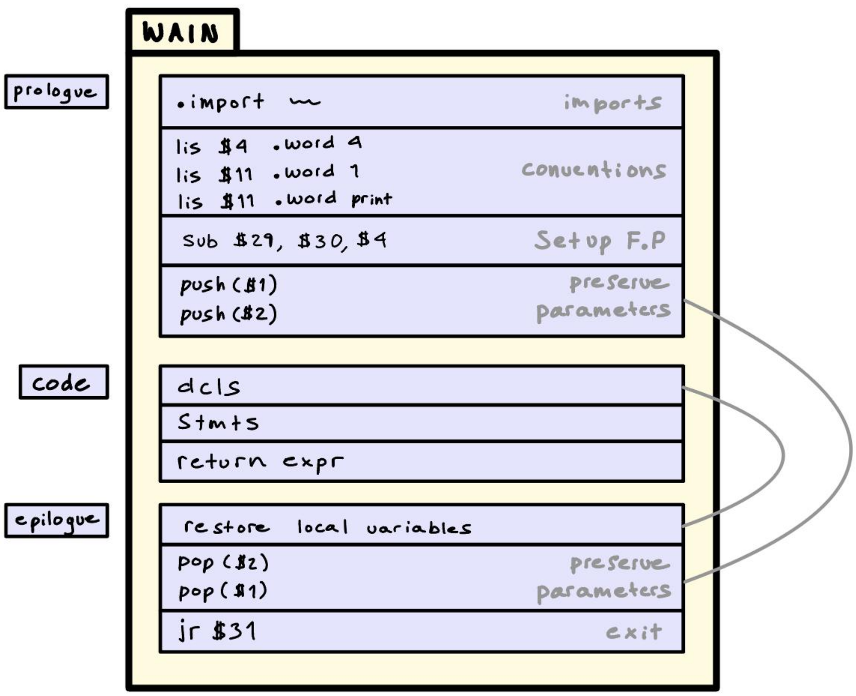
Summary

1. Compiler adds ".import println" to the top of the code
2. Compiler output: assembly + imports
3. Assembler (assembly + imports→ object)
4. Linker (object files → combined object file)
5. Generate MIPS machine code



Conventions

\$10	"print" (address)
\$11	1
\$4	4
\$5	Temporary values



Comparisons

Convention: \$3 ⇐ 1 : comparison is true

test	→	$expr_1$ * $expr_2$
code(test)	=	**code** ($expr_1$) **push** (\$3) **code** ($expr_2$) **pop** (\$5) + code for *

*	code for *
<	slt $3 ,$5 ,$3
>	slt $5, $3 ,$3
!=	slt $6 ,$3 ,$5 slt $7 ,$5 ,$3 add $3 , $6 , $7
==	code(!=) sub $3 , $11 , $3
<=	code(>) sub $3 , $11 , $3
>=	code(<) sub $3 , $11 , $3

Control-Flow

if statement

statement	→	IF (test) { stmts$_1$ } ELSE { stmts$_2$ }

code()	=	code (test)
		beq $3 , $0, *else*
		    code (stmts$_1$)
		    beq $0 , $0 , *endif*
		else:
		    code (stmts2)
		endif:

statement	→	WHILE (test) { stmts }

code()	=	loop:
		    code(test)
		    beq $3 , $0 , *endWhile*
		    code (stmts)
		    beq $0 , $0 , *loop*

		endWhile:

Note: ensure that the labels are unique across the entire output generated by the compiler

Pointers

Pointer are always initialized to **NULL**

We associate associate NULL with the address **0x1**

- not divisible by 4, leading to an unaligned address (program crash). Which is what we want (can't dereference NULL).

factor	→	**NULL**
code()	=	add $3, $0 , $11

Dereferencing

factor	→	**STAR factor**
code()	=	code (factor$_2$) ; $3 = address lw $3 , 0($3)

Get Address of

	factor	→	**AMP lvalue**
1	lvalue = ID		
	code()	=	lis $3 .word offset (from symbolTable) $3 = $29 + $3
2	lvalue = STAR factor		
	code()	=	code(factor)

Assignment Through Pointer Dereference

	statement	→	lvalue BECOMES expr SEMI
1	lvalue = ID		
	code()	=	code(expr) sw $3, offset($29)
2	lvalue = STAR factor		

	code()	=	code(expr) //value
			push($3)
			code(factor) //address
			pop $5
			sw $5, 0($3)

Comparisons

Pointer Comparison	Integer Comparison
cannot be negative	can be negative
unsigned comparison	signed comparison
sltu	slt

Problem:

Consider the comparison a < b:

- The parse tree will look identical, irrespective of the types of a and b
- Yet, the code to be generated (slt or stlu) will be different

Solution:

- During type-checking, **augment** each node with the **type** that has been **inferred** (provided the node has an associated type)
- it's enough to retrieve the type of only 1 of the operands of a comparison

Arithmetic

Note: sizeof(int) = 4

<table>
<tr><th></th><th>expr</th><th>→</th><th>expr + term</th></tr>
<tr><td rowspan="2">1</td><td colspan="3">int + int*</td></tr>
<tr><td>code()</td><td>=</td><td>expr + (4 × term):

code(expr)
push($3)
        code(term)
        mult $3, $4
        mflo $3
pop $5
$3 = $5 + $3</td></tr>
<tr><td rowspan="2">2</td><td colspan="3">int* + int</td></tr>
<tr><td>code()</td><td>=</td><td>(expr × 4) + term</td></tr>
</table>

<table>
<tr><th></th><th>expr</th><th>→</th><th>expr - term</th></tr>
<tr><td rowspan="2">1</td><td colspan="3">int* + int</td></tr>
<tr><td>code()</td><td>=</td><td>(expr × 4) - term</td></tr>
<tr><td rowspan="2">2</td><td colspan="3">int* + int*</td></tr>
<tr><td>code()</td><td>=</td><td># elements between 2 addresses:

code (expr)
push ($3)
code (term)
pop ($5)

sub $3 , $5 , $3   ; $3 = expr₂ - term
div $3 , $4   ; $3 / 4
mflo $3</td></tr>
</table>

Heap Memory Allocation

We rely on the runtime environment to provide support for **new** & **delete**

Assume the presence of: **alloc.merl**

alloc.merl

Exports 3 labels:

1. init
2. new
3. delete

addition to prologue
.import init .import new .import delete

init label

- Procedure
- Call once before any calls to new or delete
- initializes the heap allocator's internal data structures

Expects	**Returns**

Argument ⇒ $2:
• If the output program will be run using mips.array (the type of the first parameter to wain is a pointer)
    ○ $2 ⇒ length of the array
• Otherwise
    ○ $2 ⇒ 0
	Na

new

Expects	Returns
# words of memory requested ⇒ $1	If memory could be allocated: • starting address of the allocated memory ⇒ $3 Otherwise: • 0 ⇒ $3

	→	**new int [expr]**
code()	=	code (expr) $1 = $3 ;new procedure expects value in $1 push ($31) lis $5 .word *new* jalr $5 pop ($31) bne $3 , $0 , 1 ; if call succeeded skip next instruction $3 = 1 ; set $3 to NULL address if allocation fails

delete

Expects	Returns
memory address to be deallocated ⇒ $1	Na

	→	**delete [ ] expr**
code()	=	code (expr) beq $3, $11, *skipDelete* ; do NOT call delete on NULL $1 = $3 ; delete expects the address in $1 push ($31) lis $5 .word delete jalr $5 pop ($31) skipDelete :

Note: deleting NULL is not an error, and simply does nothing

Assembling and linking (with alloc.merl)

Given: output.asm is generated output

cs241.linkasm < output.asm > output.merl
cs241.linker output.merl print.merl alloc.merl > exec.merl
cs241.merl 0 < exec.merl > exec.mips

Note: the way our allocator is implemented, it must appear last in the linker command

run the program:

1. mips.twoints exec.mips
2. mips.array exec.mips

Generating and Calling Procedures

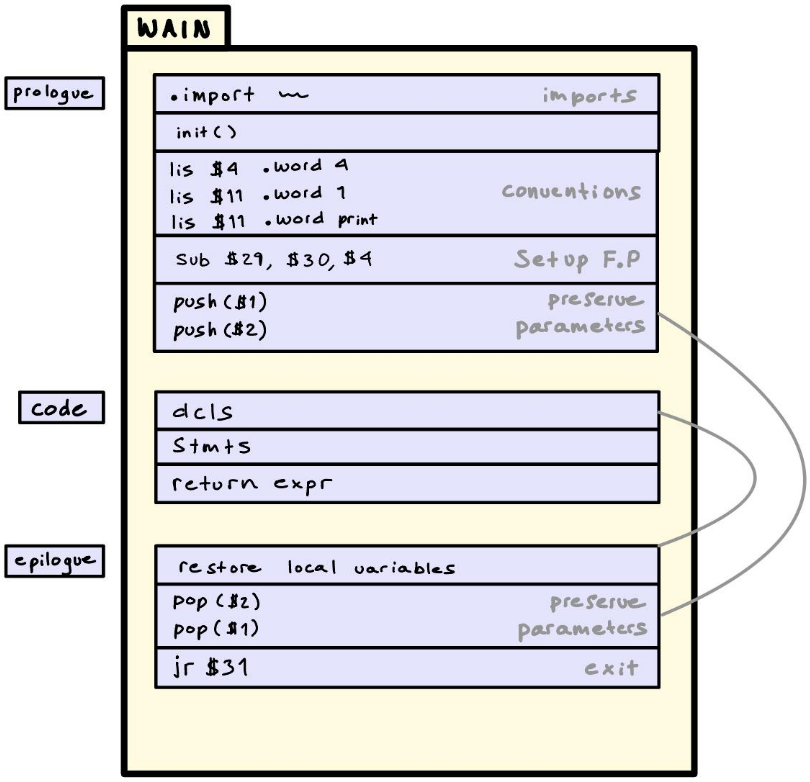
WLP4 semantics require that the program begin by executing the wain function

- Generated output for wain appears first
- Any procedures that are defined in the WLP4 program would appear after the end of the code for wain
- we must ensure that each procedure ends with the jr $31 instruction

For all procedures & wain:

1. setup frame pointer ($29)
2. Preserve parameters & local variables
3. end with jr $31

Wain



Procedures

Saving and Restoring Registers

Conventions:

$30	automatic
$31	caller (always)
$29	caller
Parameters (stack)	caller
Local Variables (stack)	callee
Other Registers	callee

$29 Caller Save	$29 Callee Save
Prologue for call: push($29) ; call procedure ($31, jalr) pop($29)	**Prologue:** push($29) ; save caller's F.P (updated $30) $29 = $30 ; set callee's frame pointer [save other registers]

Note: set frame pointer <u>before</u> saving other values to the stack

Arguments to a Procedure Call

Use the stack for passing all arguments

<u>caller saves FP & parameters:</u>

factor	→	**ID ($expr_1$, … , $expr_n$)**
code()	=	push($29) push($31) -------------------------------- code($expr_1$) push ($3) . . . code($expr_n$) push ($3) -------------------------------- lis $5 .word ID jalr $5 -------------------------------- ; pop all arguments -------------------------------- pop($31) pop($29)

Procedure Definition

We want all parameters & local-variables to occur contiguously in memory:

1. Generate code for declarations
2. Push the registers that need to be preserved

Local variable i's offset from FP = -4(i-1)

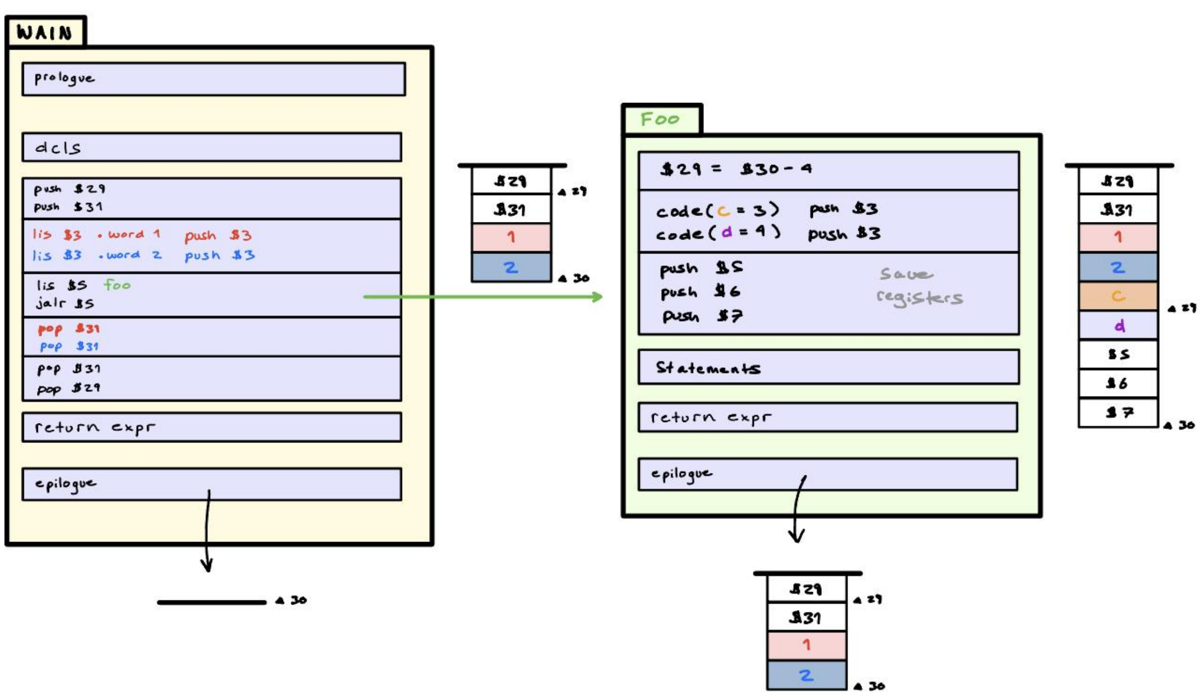
procedure	→	int ID (params) { dcls stmts RETURN expr; }

code()	=	ID:
		$29 = $30 - 4 //caller saves FP
		code(dcls) //local variables ⇒ stack
		push (registers to save) //if the callee saves registers
		code(stmts)
		code(expr) //return expr
		pop(saved registers)
		pop (local variables)
		jr $31

Example

```
int foo(int a, int b)
{
    int c = 3;
    int d = 4;
    ...
}

int wain(...) {
{
    ...
    foo(1,2);
    ...
}
```



Duplicate Labels

Our code generation scheme requires the generation of labels in certain situations

We need each label to be unique. But functions can be named after other things.

Solution:

- attach a prefix (underscore) to all WLP4 function names
    - Append, "_" in front of labels corresponding to functions
- Requires that no labels start with "_" anywhere else