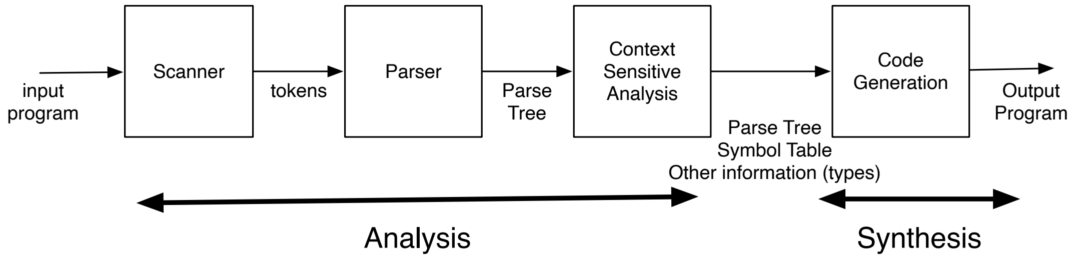
Compiled on 02/11/2022 at 9:45am

# Code Generation

In our simplistic view of a compiler, the synthesis stage comprises just of code generation. That is not entirely true of real compilers. Compilers will often perform *optimizations* both before and after the code generation stage. We discuss some optimization strategies in the next module. For now, let's talk about code generation.



The compiler gets to the synthesis stage once it has confirmed that the input program is syntactically and semantically valid, i.e., all rules required by the source programming language have been satisfied. The goal of the synthesis stage is to generate the output in some desired format. Often this is output is processor-specific assembly code, which can then be processed by an assembler to make runnable machine code. For `WLP4`, we will generate a MIPS assembly program as the output. Thus, this program can then be assembled to produce the machine code that can run on a MIPS processor.

An important distinction to make is that the synthesis stage is meant to generate a program as its output, not evaluate or execute the program. For example, if the program contains an `if` statement, the code generation stage generates an equivalent `if` statement in the target language (in our case MIPS assembly). In other words, the code generator does not need to determine whether the comparison in the `if` statement will evaluate to true or false. Of course, the compiler *could* try to optimize away a comparison if it can guarantee that a condition is always `true` or `false`. But that is an optimization and not our concern in this module; we intend to simply translate the `WLP4` program into an equivalent MIPS assembly program.

Technically speaking, there are infinitely many equivalent MIPS programs for a single WLP4 program. In this module, we spell out one way of translating the input `WLP4` program into a MIPS program. Our approach seeks out a simplistic solution which is not necessarily the most efficient. An interesting thought to ponder over is what we mean by efficiency when it comes to a compiler. A compiler can be efficient in the time it takes to compile (e.g., taking an exponentially long time in the number of lines of code to compile is likely not good). More importantly, a compiler can generate efficient code, i.e., the time it takes to execute the output generated by the compiler. While efficiency to compile is important, efficiency of the generated code is even more important. As stated earlier, in this module we use a simplistic code generation mechanism which is efficient to compile but does not necessarily produce the most efficient code. In the next module, we will touch upon techniques that could make the produced output more efficient.

## 1 Accessing Values for Variables

Let's begin by generating code for some simple WLP4 programs:

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

Recall our mips.twoints convention that registers 1 and 2 store the input values to a MIPS program and register 3 stores the return value. It is no coincidence that wain is restricted to two parameters. We will use the convention that parameter 1 of wain will be present in register 1, parameter 2 in register 2, and the value returned by a WLP4 function in register 3.

The program above returns parameter a. Since we decided that the first parameter is in register 1, we must store that value in register 3 (our return register). We can do this with the add instruction that copies the value in register 1 to 3. Of course, we will need the instruction to exit from the MIPS program as well. The following would be a valid equivalent MIPS program:

```
add $3, $1, $0
jr $31
```

Now imagine that there is a very similar WLP4 program that looks exactly like the program above but instead of returning the variable a it returned variable b. The obvious output to produce is also almost identical except that the add instruction would copy into register 3 the value from register 2 and not 1. That seems fine until you think about how the code generator is supposed to know this. More specifically, the parse trees for these two programs are essentially identical. In fact, the only difference between the two parse trees is the lexeme stored in the ID token within the expression in the return statement. This means that the structure of the parse tree alone is not going to be enough to determine what output to produce. In particular, we need a way to map where each variable is stored. Recall from our context-sensitive analysis stage that we already have a symbol table for each procedure that maps variable names to their type. We can extend this symbol table with a location entry for each variable as follows:

| Name | Type | Location |
|---|---|---|
| a | int | \$1 |
| b | int | \$2 |

Once we have this location entry for each variable, code generation should be simple: while traversing the parse tree, if we encounter a variable, we can refer to the location entry for that variable to see which register holds the value associated with this variable. For our example WLP4 programs, if we were to encounter the variable a, we would find, and therefore use, register 1. Similarly, if we were to encounter variable b, we would use register 2 as per the location entry for this variable.

Notice that so far, we have only talked about the parameters to wain. We could consider extending this approach to other variables in the function and reserve individual registers for them. But this leads to an obvious problem: there is no upper bound to the number of variables in a WLP4 function, but the number of registers in MIPS is limited. We have dealt with similar issues before: while registers are limited, we have a seemingly unlimited supply of memory on the stack. We can use the stack to store all variables including parameters for consistency of code generation.

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

Would generate the following code:

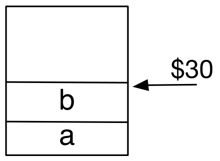
```
; begin prologue
lis $4      ; new convention $4 always contains 4
.word 4
sw $1, -4($30) ; store variable a at $30-4
sw $2, -8($30) ; store variable b at $30-8
sub $30, $30, $4
sub $30, $30, $4
; end prologue

lw $3, 4($30)   ; load from stack location reserved for variable a

; begin epilogue
add $30, $30, $4
add $30, $30, $4
jr $31
```

Before we discuss the code that was generated, notice the use of MIPS comments. These are comments that the compiler generates as part of the output it produces. These comments can be really useful when debugging the output produced by a code generator.

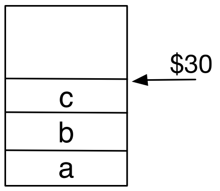
In the code we generated, instead of keeping track of which register contains the variable, we now track which location on the stack stores the variable. We can do that by mapping each variable to an offset with respect to (w.r.t.) $30. Below, the diagram on the left shows the contents of the stack (including the updated stack pointer after the `prologue` part of our generated code has executed). At that point, variable a is at offset 4 and variable b at offset 0 w.r.t. $30.



| Symbol | Type | Offset (from $30) |
|---|---|---|
| a | `int` | 4 |
| b | `int` | 0 |

Do you see a problem? First, the offset to variables will depend on the number of variables in the function. Second, the stack is likely going to be used for holding other values. Every time the stack pointer is updated, our offsets will change! The first issue is easy to resolve in `WLP4` since all variables must be declared at the top of the function. The code generator can precompute the offsets for each variable in the program and set up the stack accordingly in the `prologue`.

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



| Symbol | Type | Offset (from $30) |
|---|---|---|
| a | `int` | 8 |
| b | `int` | 4 |
| c | `int` | 0 |

Using this scheme, our code generator can correctly generate the following output:

```
; begin prologue
lis $4
.word 4
sw $1, -4($30)
sw $2, -8($30)
sub $30, $30, $4
sub $30, $30, $4
sub $30, $30, $4    ; space for c
```

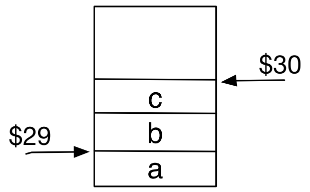
```
sw $0, 0($30)       ; c = 0
; end prologue

lw $3, 8($30)       ;8 from the symbol table

; begin epilogue
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

While this fixes the first issue we identified, i.e., needing to know the offset for all variables, we have the second concern to deal with, i.e., the stack pointer will likely be updated once we start using the stack to store additional information, e.g., making procedure calls. A standard approach is to dedicate another register to always point to a fixed memory location within the stack while executing a procedure. We call this a frame pointer. By keeping the frame pointer unchanged within a procedure, we can use fixed offsets w.r.t. the frame pointer. We will use $29 as the frame pointer. The choice of what we initialize $29 to is up to us and is immaterial as long as it is done consistently. We will initialize $29 to the first value in the frame for a procedure.

```
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 |

Notice how this avoids needing to update offsets within our code generator when we create code that updates the stack pointer. Since all variables are referenced with respect to $29, the frame pointer, changes to the stack pointer have no effect on the offsets. The following code uses the new code generation strategy:

```
; begin prologue
lis $4
.word 4
sub $29, $30, $4 ; setup frame pointer
sw $1, 0($29)    ; push variable a
sub $30, $30, $4 ; update stack pointer
sw $2, -4($29)   ; push variable b
sub $30, $30, $4 ; update stack pointer
sw $0, -8($29)   ; push variable c
sub $30, $30, $4 ; update stack pointer
; end prologue

lw $3, 0($29)    ; variable a has offset 0 w.r.t. $29

; begin epilogue
add $30,$30,$4
add $30,$30,$4
add $30,$30,$4
jr $31
```

We will revisit this decision at the very end of this module where we discuss procedure calls.

## 2 Generating code for expressions

In the previous section, we generated code for simple expressions, just the variable `a` or `b` or `c`. Let's now look at more complicated expressions:

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

Before we continue, let's define shorthand for our code generator. We will write `code(a)` to indicate the code that our code generator will generate to load the value for variable `a` into register $3. For the example above, since variable `a` is parameter 1 and is at offset 0 w.r.t. $29, `code(a)` will be `lw $3, 0($29)`. Later, we will generalize `code` for other rules in our grammar.

Going back to the example above. This isn't much more complicated than before. Since the return value is to be stored in register 3, we can load the value for variables `a` and `b` into two registers, perform the subtraction and ensure that the result is in $3. However, the following is obviously not going to work:

```
code(a) ; results in lw $3, 0($29), which we will write as $3 <- a
code(b) ; results in lw $3, -4($29), which we will write as $3 <- b
sub .....
```

Notice the obvious issue: loading the value for `b` in $3 clobbers[1] the previous value in that register (the value for variable `a`). We need a temporary. In pseudo-code this could look as follows:

```
code(a)      ; $3 <- a
add $5,$3,$0 ; $5 <- $3
code(b)      ; $3 <- b
sub $3,$5,$3
```

In the pseudo-code above, we use $5 as a temporary register. This pseudo-code would generate the following snippet of output:

```
lw $3,0($29)  ; a is at offset 0 w.r.t. $29
add $5,$3,$0  ; Store a in $5
lw $3,-4($29) ; b is at offset -4 w.r.t. $29
sub $3,$5,$3
```

We needed one extra register to use as a scratch/temporary register for the expression `a-b`. Unfortunately, the approach does not scale for arbitrarily complicated expressions. For example, the following is the pseudo-code for the code generator for the expression `a+(b-c)` and requires two temporary registers:

```
code(a)      ; $3 <- a
add $5,$3,$0 ; $5 <- $3
code(b)      ; $3 <- b
add $6,$3,$0 ; $6 <- $3
code(c)      ; $3 <- c
sub $3,$6,$3 ; $3 <- b-c
add $3,$5,$3 ; $3 <- a+(b-c)
```

[1] "Clobber" is actually the term of science here. It is the standard term for one step overwriting the result of a previous step due to using the same storage, particularly a CPU register.

We needed two temporary registers since we had to temporarily store the result of both `a` and `b`. We are in a familiar situation: the limited nature of registers. And we will use a familiar solution: store temporary values on the stack. Let's define more shorthand. We will write `push($3)` to represent a store of the value in the supplied register (in this case register 3) on the stack and an appropriate update to the stack pointer. We will also write `pop($5)` to represent a load from the top of the stack into the supplied register (in this case $5) and an appropriate update to the stack pointer.

```
push($3) generates:
sw $3,-4($30)
sub $30,$30,$4
```

```
pop($5) generates:
add $30,$30,$4
lw $5,-4($30)
```

Note the convention that $4 always contains the value 4.

The pseudo-code for generating the code for `a-b` can be written as:

```
code(a)
push($3)
code(b)
pop($5)
sub $3,$5,$3
```

Notice that this produces suboptimal code; why push $3 when we are going to pop it soon after? The intent here is that of simplicity, and perhaps more importantly, universality: this would work no matter how complicated `code(b)` is, so long as it puts its result in $3 and is consistent with the stack. This is a universal code generation scheme for all expressions.

Write, in pseudo-code just introduced, the code generated for the following program:

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

The answer to the above is shown below.

```
; begin prologue
lis $4
.word 4
sub $29, $30, $4
push($1)
push($2)
lis $5
.word 3
push($5)
; end prologue

code(a)             ;Load a in $3
push($3)
code(b)             ;$3 <- b
push($3)
code(c)             ;$3 <- c
pop($5)             ;$5 <- b
sub $3, $5, $3      ;$3 <- b-c
pop($5)             ;$5 <- a
add $3, $5, $3      ;$3 <- a+(b-c)

; begin epilogue
```

```
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

The above coding strategy uses one temporary register (we chose \$5) to compute any arbitrarily complicated expression. We will continue to create shorthand to represent our coding strategies. Let's now extend our shorthand for rules in the grammar. For example, consider the grammar rule $\texttt{expr}_1 \rightarrow \texttt{expr}_2$ `PLUS term`, where the numbers 1 and 2 have been added for illustrative purposes to distinguish the two expression non-terminals. In the parse tree, this would be represented by a node for $\texttt{expr}_1$ which would have three children, the node for $\texttt{expr}_2$, the node for `PLUS` and the node for `term`. We will write the following shorthand for generating the code for $\texttt{expr}_1$:

```
code(expr₁) = code(expr₂) + push($3)
            + code(term) + pop($5)
            + add $3, $5, $3
```

Since other arithmetic operations on integers (subtraction, multiplication, division and modulo) work the same way as addition, we will not give pseudo-code for them. We discuss pointer arithmetic later.

Based on this shorthand, we give code generation strategies for other rules in the grammar. Some rules are trivial to specify so we handle them first:

For the rule `S` $\rightarrow$ $\vdash$ `procedure` $\dashv$: `code(S) = code(procedure)`

Similarly, for the rule `expr` $\rightarrow$ `term`: `code(expr)` = `code(term)`

Also, `factor` $\rightarrow$ `LPAREN expr RPAREN`, we have `code(factor) = code(expr)`.

While we will cover the assignment statement in full detail later, let's look at one case for when a variable is assigned a value. The rule is `statement` $\rightarrow$ `lvalue BECOMES expr SEMI` when `lvalue` $\rightarrow$ `ID`. The code to generate in pseudo-code is as follows:

```
code(statement) = code(expr)
                + sw $3, offset($29)
```

where `offset` is the offset value computed by the compiler. Recall, we are covering the case where the Left-Hand-Side is an `ID`, i.e., a variable. We have already decided that the compiler will maintain a location for each variable as an offset w.r.t. register 29. The compiler chooses the appropriate offset value by looking up the offset assigned for the variable for this `ID`.

## 3 Generating Code for Print Statement

The `WLP4` language provides a print statement through the rule:

```
statement → PRINTLN LPAREN expr RPAREN SEMI
```

The language defines the semantics: the expression is evaluated, and its value is printed to standard output followed by a newline. Also, from our context-sensitive analysis, you will recall that the expression must be an integer, i.e., we can only print `int` values [2].

[2] You may recall that you implemented this exact behaviour in a MIPS assembly program in an assignment.

We can handle this in three ways:

1. The compiler can output the full code to convert a binary number to decimal and to ASCII and to output it every time `prinln` is used.

2. The compiler can output a `print` procedure **once** and then call this procedure every time a `println` statement is used.

3. The compiler can **import** a `print` procedure and assume that this procedure is available in the *runtime environment*. The compiler then calls this **imported** procedure every time a `println` statement is used.

The first and second of these approaches are nothing new. The third introduces the concept of runtime environments.

**Definition 1** *A runtime environment is the execution environment provided to an application or software by the operating system to assist programs in their execution. Such an environment may include things such as procedures, libraries, environment variables and so on.*

For example, the standard C library, providing functions such as `malloc`, `free`, `printf` and `memcpy`, is part of the execution environment of C programs. On Unix and Unix-like systems including Linux, it is provided in `libc.so`. Archaic, nonstandard operating systems have other conventions; for instance, on Windows, it is provided by `msvcrt.dll`.

Real compilers rely on runtime environments since this allows the binary they produce to be smaller. Additionally, there is no need to generate the same code again and again if every (or many) programs that are compiled rely on the same code. Instead, the approach taken is to make the common code available as part of the runtime environment. The pre-compiled code in the runtime environment must then be *linked* with the output that the compiler produces which assumes the presence of certain code in the runtime environment. This linking process produces the standalone executable. Below, we discuss, at a high level, how we can use `print` from the runtime environment. We will take on a more detailed discussion of linking in future modules.

The pre-compiled code in the runtime must contain information regarding what the code provides and what it expects. This information goes beyond the target language. For example, if our runtime environment is to support a `print` procedure, not only must the pre-compiled version contain the MIPS machine code for the `print` procedure, it must also contain some way for this information to be "announced". Similarly, the output that the compiler generates, that assumes the presence of the `print` procedure, must now announce that the presence of `print` was assumed. This is the concept of *object files* that you might be familiar from earlier courses such as CS246. An object file, apart from containing the compiled version of some code, contains additional information regarding what a piece of code expects and what it provides.

In CS241, we use our homegrown *MIPS Executable Relocatable Linkable (MERL)*[3] format for object files. The `MERL` format contains MIPS machine code (not assembly) along with additional information needed by both the *linker* and the *loader*. We will be addressing these topics and the format

[3]MERL is based very loosely on the so-called "a.out" format, which was the original format for object and executable files on Unix. That format actually has no name, and was simply called "a.out" retrospectively because this is the default name for the output of Unix C compilers.

in a lot more detail in future modules.

While generating code, if the compiler needs to use the `print` procedure that will be part of the runtime, it will generate an assembler directive to `import` print. Importing a procedure named `print` simply requires having the following appear **once** in the generated assembly (by convention at the top of the generated file):

```
.import print
```

Once the compiler is done generating output, the result is an assembly file which might now contain `import` statements. This means that we must now use an assembler that understands imports. We call this assembler the *Linking Assembler* (`cs241.linkasm`). It is an assembler that produces object files rather than just MIPS machine code. Suppose the output generated by the compiler is in a file named `output.asm`. Running `cs241.linkasm` produces a `MERL` file:

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

The `output.merl` file is the object file for the `WLP4` program the compiler was to compile. It contains MIPS machine code along with an announcement that this code expects to be linked to an appropriate `print` procedure. Imagine the existence of `print.merl`, an object file for the `print` procedure. We can link them using a linker that understands the `MERL` format:

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

Notice how the linker takes a number of object files, links them, and produces a new object file which contains the combined machine code and "announcements". To produce the pure MIPS machine code, we must strip out the MERL metadata from the object file. This can be done using the `cs241.merl` tool. We show how to use the tool below and will discuss this in more detail in a future module:

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

Coming back to the topic at hand, we can generate code for the `WLP4` `println` statement by relying on our knowledge of what the `print` procedure we will be linking with expects. This procedure expects the integer to be printed in register 1. Just like any other procedure, we will call print using the `jalr` instruction which overwrites register 31. Using the shorthand we had previously developed, we can write the following:

```
code(println(expr);) = push($1)  ; if current value in $1 needs to be preserved
                     + code(expr)
                     + add $1, $3, $0
                     + push($31)
                     + lis $5 + .word print
                     + jalr $5 + pop($31)
                     + pop($1)
```

In the pseudo-code above, we chose to save the original value in register 1 before copying into it the value to be printed. This might be un-necessary depending on other decisions made during code generation. Notice also, how the pseudo-code above uses register 5 to hold the address of the `print` label that is assumed to be present. This could be made part of a convention, e.g., say register 10 could be dedicated to always hold the address of `print` much like how we have dedicated register 4 to always hold 4.

## 4 Summarizing conventions so far

The following code describes some of the conventions we have chosen. We have added the convention that \$11 will always hold the value 1 since that is a value that is often used for comparisons which we talk about in the next section.

```
; begin Prologue
.import print
lis $4     ; $4 will always hold 4
.word 4
lis $10    ; $10 will always hold address for print
.word print
lis $11    ; $11 will always hold 1
.word 1
sub $29, $30, $4    ; setup frame pointer
; reserve space for variables

; end Prologue and begin Body

; translated WLP4 code

; end Body and begin Epilogue

; deallocate parameters and local variables of wain

jr $31
```

The only other convention that is not apparent from the snippet above is that while evaluating any arbitrary expression, we use register 5 to temporarily hold values.

## 5 Comparisons

Recall that the `WLP4` grammar has a number of rules for the non-terminal `test` that appear as the condition in the `if` and `while` control-flow statements. We discuss the generation of assembly code that would evaluate these conditions. It is important to remember that we are not evaluating the condition; we are simply generating the equivalent assembly code for the `WLP4` condition.

For the rules `test` $\rightarrow$ $\texttt{expr}_1 < \texttt{expr}_2$ we generate:

```
code(test) = code(expr₁)
           + push($3)
           + code(expr₂)
           + pop($5)
           + slt $3,$5,$3
```

Recall that the shorthand code(...) generates the code for a component such that when the generated code executes the result will be in register 3. The code above generates the code for $\texttt{expr}_1$ with the result being placed in register 3. Then, this value is pushed onto the stack. Then the code for $\texttt{expr}_2$ is generated which, when executed, will place the value for $\texttt{expr}_2$ in register 3. At this point, we retrieve the value for $\texttt{expr}_1$ by popping the top of the stack into register 5. The comparison then compares the two expressions, placing the result of the comparison in register 3, as per our convention. Later, we will be able to check whether the condition was true; register 3 is 1 if $\texttt{expr}_1$ is less than $\texttt{expr}_2$, i.e., our convention is to generate code which will cause the value in

$3 to be 1 whenever a comparison is true.

Generating the code for `test` → $\texttt{expr}_1$ > $\texttt{expr}_2$ can reuse the compiler's code for the expression above; we simply switch the order for where we generate the code for $\texttt{expr}_1$ and $\texttt{expr}_2$, i.e., checking that $\texttt{expr}_1$ > $\texttt{expr}_2$ is the same as checking that $\texttt{expr}_2$ < $\texttt{expr}_1$.

Let's now generate the code for `test` → $\texttt{expr}_1$ ! = $\texttt{expr}_2$. There are a number of ways of achieving this. One way is to use the fact that if $\texttt{expr}_1$ is indeed not equal to $\texttt{expr}_2$, then either $\texttt{expr}_1$ is less than $\texttt{expr}_2$ or $\texttt{expr}_2$ is less than $\texttt{expr}_1$.

```
code(test) = code(expr₁)
           + push($3)
           + code(expr₂)
           + pop($5)
           + slt $6,$3,$5 ; $6 = $3 < $5
           + slt $7,$5,$3 ; $7 = $5 < $3
           + add $3, $6, $7
```

After generating the code so that the evaluated values of $\texttt{expr}_1$ and $\texttt{expr}_2$ are in registers 5 and 3 respectively, we perform comparisons that check whether $\texttt{expr}_2$ is less than $\texttt{expr}_1$ and vice versa. The results of these comparisons are in registers 6 and 7 respectively. If either one of these comparisons is true (i.e., if either of register 6 or 7 are one) then the condition is true. One way of achieving this is to simply add the values in registers 6 and 7. The result will be 1 only if one of the conditions was true and 0 if both conditions are false.

Generating the code for equality, `test` → $\texttt{expr}_1$ == $\texttt{expr}_2$, can use the code for not equal by using the idea that a==b is the same as !(a !=b). This can be achieved simply by flipping the value in register 3 as computed by the code above, i.e., we generate the code for != and then add the line

```
sub $3, $11, $3
```

Recall that we have said that $11 always contains the value 1. So, if $3 was 1, it becomes 0 and if it was 0, it now becomes 1.

The comparisons that we have not discussed so far are <= and >=. Once again, we can prudently reuse the code we already know how to generate. This can be done by noticing that $a <= b$ is the same as $!(a > b)$ and similarly $a >= b$ is the same as $!(a < b)$.

## 6 Control-Flow Statements

We are left with the two control flow statements: `if` and `while`. Let's look at the rule for the `if` statement first: `statement` → `IF` (`test`) {`stmts1`} `ELSE` {`stmts2`} . In the previous section, we talked about the pseudo-code, code(`test`), which, given any `test`, will generate Assembly code such that $3 will be 1 if the test is true and 0 if false. Assuming that the shorthand `code(statements)` generates code for any sequence of statements, we can define the following pseudo-code for an `if` statement:

```
code(statement) = code(test)
                + beq $3, $0, else
                + code(stmts1)
                + beq $0, $0, endif
                + else:
                + code(stmts2)
                + endif:
```

Notice the use of the labels `else` and `endif`. If the test condition is false, i.e., $3 is 0, we must execute the statements in the `else` block. We generate the `beq $3, $0, else` instruction to jump to that block. Otherwise, we generate the code for the `true` block. Of course, if the true block is executed, at the end, we must skip over the false block; hence the `beq $0, $0, endif`.

While the code above works for an `if` statement, we must ensure that the labels we use are unique across the entire output generated by the compiler. In particular, we cannot use exactly the same label when generating code for multiple `if` statements. One simple way is to append to each label a counter that is incremented every time a new label is generated.

Generating the code for `while` statements is similar. For the rule `statement` → `WHILE` (`test`) {`statements`} we can generate the code as shown in the following pseudo-code:

```
code(statement) = loop:
                + code(test)
                + beq $3, $0, endWhile
                + code(stmts)
                + beq $0, $0, loop
                + endWhile:
```

Once again, it is important to ensure that labels being generated are unique to the entire output.

## 7 Pointers in WLP4

Until now, we have ignored the presence of the pointer type in `WLP4`. In this section, we discuss how our code generation strategy needs to differ when dealing with pointers.

### 7.1 Null Values

Pointer variables are initialized to the special `NULL` value. From C, we are used to thinking of `NULL` as the address `0x0`. This works for languages like C since the address `0x0` is typically the starting address of the `code` segment of the program and the operating system prevents the program from accessing that segment, leading to the familiar *segmentation faults*[4]. In MIPS assembly, `0x00` is a valid address (if the loader loaded the program starting at that address, as per our ongoing assumption, `0x00` is the first line of the program). This means that accessing the address `0x0` would not result in an error. But we want it to, since that is the behaviour you get from dereferencing `NULL`. One way to achieve the desired behaviour is to associate `NULL` with an address that is not divisible by 4. For instance, we can choose `0x1`. Recall that if a MIPS program tries to access memory at an unaligned address, i.e., an address that is not a multiple of 4, it causes the program to crash. Since that is exactly the behaviour we want, we will associate `NULL` to `0x1`.

[4]You'll learn more about how and why the operating system does this in CS350.

The actual code is trivially simple to generate, `factor` → `NULL` would lead to:

```
code(factor) = add $3, $0, $11 ; recall $11 is always 1
```

With this behaviour, anytime the program tries to dereference this address (with either `lw` or `sw`) it will result in a crash since MIPS is expecting a word-aligned address.

## 7.2 Dereferencing Pointers

Pointer variables are dereferenced through the rule $\texttt{factor}_1$ → STAR $\texttt{factor}_2$ (recall we add the subscripts to distinguish between different occurrences of the same non-terminal). Since the input already passed through the type-checking phase, we know that $\texttt{factor}_2$ must evaluate to a pointer type (since we can only dereference pointers). This rule appears when we are dereferencing a pointer to access the value stored at that address; there is a separate rule for when we want to access the location to write to the address which we discuss later. Since $\texttt{factor}_2$ is an address, and we want to load from that address, we will use the `lw` instruction:

```
code(factor1) = code(factor2) ; $3 will contain the address
              + lw $3, 0($3)  ; $3 contains the loaded value
```

## 7.3 Taking the Address-Of

The address-of operator & is used to obtain a pointer to a memory location. This means that you can only take the address of an `lvalue` since, by definition, `lvalues` represent a storage location. The `WLP4` grammar ensures that the address-of operator is only syntactically valid for an `lvalue` via the rule: `factor` → `AMP lvalue`. What would have been harder is the case if the rule for address-of looked like $\texttt{factor}_1$ → AMP $\texttt{factor}_2$. In that case, since for example $\texttt{factor}_2$ could derive a NUM, the compiler would need to go to extra lengths to ensure that the operand to the address-of operator is an lvalue (it does not make sense to be taking the address of a number).

In the grammar rule for the address-of operator, `factor` → `AMP lvalue`, the `lvalue` can derive an `ID` or `STAR factor`[5]. Let's consider both cases. In the case that `lvalue` → `ID`, the intent is to get the address of a variable. Recall our decision to store all variables on the stack at offsets with respect to the frame pointer, $29. We have these offsets recorded in the symbol table. Therefore, we can use this information to load the address allotted to the variable. The pseudo-code for `factor` → `AMP lvalue` when `lvalue` → `ID` can be written as:

```
code(factor) = lis $3
             + .word offset
             + add $3, $3, $29
```

where `offset` is the offset found in the symbol table for `ID`.

Let's look at the other case, i.e., when $\texttt{factor}_1$ → AMP lvalue and lvalue → STAR $\texttt{factor}_2$. In other words, the expression is & ∗ $\texttt{factor}_2$, i.e., first dereference $\texttt{factor}_2$ and then take the address of that location. The two operators simply cancel each other out, so the code for $\texttt{factor}_1$ is just the code for $\texttt{factor}_2$:

```
code(factor₁) = code(factor₂)
```

[5]Technically, there is a third case where the `lvalue` derives an `lvalue` wrapped in parentheses. That case degrades to dealing with the inner `lvalue`.

## 7.4 Assignment Through Pointer Dereference

The relevant rule is `statement` → `lvalue BECOMES expr SEMI`. Once again, we consider the two cases for `lvalue`: when `lvalue` → `ID` and when `lvalue` → `STAR factor`. The first of the two cases, when `lvalue` is an `ID`, was discussed much earlier and does not deal with assignment through pointer dereference. Nevertheless, we repeat the pseudo-code below for ease of reference:

```
code(statement) = code(expr)    ; $3 <- expr
                + sw $3, offset($29)
```

where `offset` is the offset for the `ID` that is `lvalue`.

Let's talk about the other case, when the left-hand side is of the form `STAR factor`, i.e., a value is being assigned to a location through a pointer dereference, e.g., `*foo = 5;`. The reasoning is straightforward. First, since the program has passed through type-checking, we know that the `factor` is a pointer, i.e., for our example, we know that `foo` has type `int*`. Since this is a pointer, our pseudo-code `code(factor)` will generate an address in $3. We must store the new value at that address. This gives us:

```
code(statement) = code(expr)   ; generate code for right-hand side
                + push($3)
                + code(factor) ; generate code for left-hand side
                + pop($5)      ; $5 contains value to assign
                + sw $5, 0($3)
```

## 7.5 Revisiting Comparisons

We already discussed comparisons (tests) used for control-flow statements. However, we did not consider pointer types when creating our pseudo-code that generates output. Recall that our type-checking stage already guarantees that the types of the two expressions being compared are the same. The code to generate for when comparing pointers is almost identical for when comparing integers with one exception: since pointers cannot be negative, the comparison should be performed using unsigned comparison, i.e., instead of using `slt` we should use `sltu`.

Until now we had been able to rely on rules we stored within parse tree nodes to decide what code to generate. This meant that even though we could have been dealing with expressions that are pointers, we did not have to look up whether they were pointers. For example, in the previous section we discussed generating code for assignment through pointer dereference. We never actually had to look up a type; the fact that the left-hand side used a STAR operator guaranteed that we were dealing with a pointer dereference. Our discussion of generating different code for comparisons based on the type of the expression confronts us with our first situation where we need to look up the type of the expression. For example, consider the comparison `a` < `b`. The parse tree (and the rules within the parse tree) will look identical, irrespective of the types of `a` and `b`. Yet, the code to be generated must choose to use `slt` if the types are integers and `stlu` if the variables being compared are pointers.

The information we need at this point must have been computed during the type-checking phase. The compiler writer has two options: (1) repurpose the type-checking analysis to re-run the code that determines what the type of an expression will be, or (2) during type-checking, augment each node with the type that has been inferred (provided the node actually can have an associated type).

Option 2 is preferred since it avoids repeating work that had already been done. Notice that it is enough to retrieve the type of only *one* of the operands of a comparison; the fact that we have passed through type-checking guarantees that this type will match the type of the other operand.

## 7.6 Revisiting Arithmetic

Earlier we discussed how we would evaluate arbitrarily complex expressions. However, we had confined ourselves to performing arithmetic on integer values. `WLP4` does support pointer arithmetic. In this section, we discuss the code that must be generated for such arithmetic.

Our first case is the rule $\texttt{expr}_1 \to \texttt{expr}_2 + \texttt{term}$ where $\texttt{type}(\texttt{expr}_2) == \texttt{int} *$ and `type(term) == int`, i.e., adding an integer value to an address. When we add an integer to an integer pointer, we do not simply add the integer value, we add `sizeof(int)` times the integer value. For example, if we had the address of the first element of an array and we added the integer 3 to it, we are hoping to compute the address of the fourth element. This fourth element is at an offset of $3 * \texttt{sizeof(int)}$ from the address of the first element. Recognizing this fact about pointer addition, we must compute $\texttt{expr}_2 + (4 \times \texttt{term})$, since `sizeof(int)` is 4, for which we write the following pseudo-code:

```
code(expr₁) = code(expr₂)
            + push($3)
            + code(term)
            + mult $3, $4 ; $4 always has the value 4
            + mflo $3
            + pop($5)      ; $5 <- expr₂
            + add $3, $5, $3
```

The case where $\texttt{expr}_1 \to \texttt{expr}_2 + \texttt{term}$ where $\texttt{type}(\texttt{expr}_2) == \texttt{int}$ and $\texttt{type}(\texttt{term}) == \texttt{int}*$ is essentially the same as the case we just covered, only we switch the order of the operands, i.e., we compute $(\texttt{expr}_2 \times 4) + \texttt{term}$.

Our next interesting case is pointer subtraction. Consider the case where $\texttt{expr}_1 \to \texttt{expr}_2 - \texttt{term}$ where $\texttt{type}(\texttt{expr}_2) == \texttt{int}*$ and $\texttt{type}(\texttt{term}) == \texttt{int}$. The code to generate is essentially the same as that from above, only instead of adding we will be subtracting to compute $\texttt{expr}_2 - (4 \times \texttt{term})$.

Perhaps the more interesting case is $\texttt{expr}_1 \to \texttt{expr}_2 - \texttt{term}$ with both $\texttt{type}(\texttt{expr}_2) == \texttt{int}*$ and $\texttt{type}(\texttt{term}) == \texttt{int}*$. Subtracting one address from another is allowed as it gives an easy way to determine the number of elements (in this case integers) between these two addresses. This can be computed by subtracting the value of `term` from $\texttt{expr}_2$ (which gives the raw memory address difference) and then dividing by `sizeof(int)` to get the number of elements, i.e., $(\texttt{expr}_2 - \texttt{term})/4$. We write the following pseudo-code:

```
code(expr₁) = code(expr₂)
            + push($3)
            + code(term)
            + pop($5)       ;$5 <- expr₂
            + sub $3, $5, $3
            + div $3, $4
            + mflo $3
```

## 8 Heap Memory Allocation

WLP4 does support heap memory. The new expression is meant to allocate a block of memory and the delete statement deallocates blocks of allocated memory. Much like what we did for println, we will rely on the runtime environment to provide support for allocation and deallocation of memory. In particular, we will assume the presence of an alloc.merl module which we now discuss. The alloc.merl module is assumed to export three labels: init, new and delete. The compiler can utilize these labels by importing them via the assembler .import directive like we did for print.

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

The init label is a procedure that must be called once before any calls to new or delete. The procedure initializes the heap allocator's internal data structures (we will discuss this later in the course). The procedure expects an argument in \$2:

- If the output program will be run using mips.array, then \$2 is the length of the array. Recall that you can check whether the WLP4 program is intended to be run using mips.array by checking the type of the first parameter to wain; if it is a pointer, the program is intended to be run by mips.array.
- Otherwise, \$2 must be set to 0.

We will use the new procedure exported by the allocator module to request allocation of memory. The procedure expects the number of words of memory being requested in \$1 and returns the starting address of the allocated memory in \$3. If memory could not be allocated, \$3 is set to 0. Based on this, we can write our pseudo-code for whenever we see a new expression in WLP4 code to be:

```
code(new int [expr]) = code(expr)
            + add $1, $3, $0    ; 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
            + add $3, $11, $0  ; set $3 to NULL if allocation fails
```

To deallocate heap memory, WLP4 provides the delete statement. The allocator provides a corresponding delete procedure which requires that \$1 contain the memory address to be deallocated. The pseudo-code to generate the output is as you would expect:

```
code(delete [] expr) = code(expr)
            + beq $3, $11, skipDelete   ; do NOT call delete on NULL
            + add $1, $3, $0   ; delete expects the address in $1
            + push($31)
            + lis $5 + .word delete
            + jalr $5 + pop($31)
            + skipDelete:
```

Notice the explicit check to ensure that we do not try to call delete on NULL. As it is, we are using the value 1 to represent NULL. Calling the deallocation procedure with the address 1 is likely going to cause a serious issue. Note that in WLP4, like in C++ (and C with free), deleting NULL is not an error, and simply does nothing.

Of course, much like what we did for the labels used by the code for `if` and `while` statements, we need to guarantee that the generated label is unique.

We summarize next how we would go about assembling and linking the generated output:

```
; assume 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
```

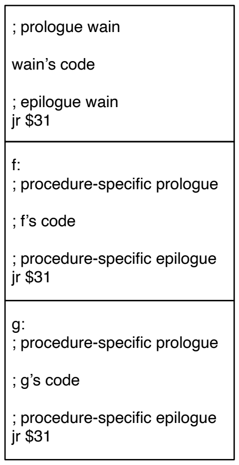
Note that the way our allocator is implemented, it must appear last in the linker command.

Then finally, we can run the generated program:

```
mips.{twoints,array} exec.mips
```

## 9 Generating and Calling Procedures

We are left with discussing how to generate the code for procedures in a `WLP4` program. Recall that `WLP4` semantics require that the program begin by executing the `wain` function. In MIPS assembly, execution begins at the first assembly instruction in the generated file. We have two options. The first is to insert instructions at the beginning of the generated program that perform any necessary initializations and then jumps to a label that represents the code produced for `wain`, e.g., `beq $0, $0, wain`, assuming that the code for `wain` is prefixed with this label. An alternate approach, which we take, is to organize our generated output such that the output for `wain` appears first. Any procedures that are defined in the `WLP4` program would appear after the end of the code for `wain`. To ensure that the code for `wain` (and other procedures) does not "fall through" to the code for the next procedure, we must ensure that each procedure ends with the `jr $31` instruction. The figure on the right shows how we can organize the generated assembly code. Notice how each procedure (other than `wain`) is simply an assembly label with that name.



```
; prologue wain

wain's code

; epilogue wain
jr $31
```

```
f:
; procedure-specific prologue

; f's code

; procedure-specific epilogue
jr $31
```

```
g:
; procedure-specific prologue

; g's code

; procedure-specific epilogue
jr $31
```

Some things, such as setting up the frame pointer ($29 as per our convention) and organizing parameters and local variables are going to be common for all procedures (including `wain`). We discuss some differences in the next two sections before going on to discuss other things such as the need to preserve register values.

### 9.1 The `wain` Procedure

The prologue for `wain` is different than the prologue for other procedures. That is because this, being the start of the program, is the place to import any external procedures (e.g. print, new etc.) and initialize any conventions (e.g. $4, $11). It also makes sense to call `init` to set up the heap memory allocator. Recall, this has to be done just once. Also, recall that $1 and $2 contain the inputs to the program. It is likely a good idea to store this (on the stack) before overwriting the values in these registers.

The epilogue for `wain` is straightforward; reset the stack (technically not needed) and then exit using `jr $31`.

## 9.2 Other Procedures

While procedures other than `wain` do not need to import external procedures, they do need to set up the frame pointer and save any registers they use in the prologue. Unlike `wain`, other procedures will need code in the epilogue to restore registers and the stack before exiting with a `jr $31`.

## 9.3 Saving and Restoring Registers

Recall our lengthy discussion in the module on MIPS assembly language regarding saving and restoring register values when calling procedures. In that module, our approach was that the **caller**, the function that was calling a procedure, saved $31. Saving all other registers was the responsibility of the **callee**, the procedure that was called. An alternate approach, where the caller saves all registers is also possible. Each approach has its own advantages and disadvantages. For example, in the caller-save approach, the caller knows which registers it actually cares about so it only saves the values that matter. However, it might end up saving registers that were never going to be over-written by the callee. On the flip side, the callee knows exactly which registers it overwrites and can therefore save just those. But then, if the caller does not care about those values, the callee is needlessly saving some of these values.

The other question is which registers do we actually need to save. The naïve approach of saving all registers is quite inefficient, especially given that the way we have proposed to generate code in this module, we only ever change values in registers $1 through $7 (while registers $4, $10 and $11 are used, they are set once at the start of the program and then never changed). Of course, $29, $30 and $31 are also changed. Of these registers, $30 is updated as needed and as long as we ensure that the callee leaves $30 where it was when it was called, the caller should be okay to proceed. For $31, of course caller-save is the way to go since the caller knows which address it must return to and the act of calling the callee (using `jalr`) will overwrite this value. The only decision to be made is that for the frame pointer, $29. We consider both approaches:

**Callee saves register 29:** In this approach the **callee** saves the old value of $29. Another decision to make is what $29 actually refers to for a procedure. The decision is up to the compiler; we use the convention that a procedure initializes its value for the frame pointer after it has stored the old value of $29, i.e., the frame pointer contains the address of the first value pushed on the stack by the callee. Based on this, we get the following prologue for a callee:

```
; prologue for call
push($29)         ; save caller's frame pointer
                  ; recall that push updated $30
add $29,$30,$0    ; set callee's frame pointer

; code to save other registers if using callee saves
```

We make the argument that setting the callee's frame pointer before saving any other registers is convenient since doing it in the other order requires keeping track of how many registers are being saved to ensure that the frame pointer is initialized to the start of the callee's frame.

The callee's epilogue would reverse what we did in the prologue. First restore any registers that were saved, then restore the caller's frame pointer by popping from the stack.

**Caller saves register 29:** An alternate, equally appealing strategy is to let the caller save its value of the frame pointer (similar to how the caller already saves the return address, $31), i.e., caller saves its frame pointer within its own stack frame. This entails that for each call the caller makes, we would need to generate the following code:

```
push($29)  ; save callers frame pointer
push($31)  ; save callers return address
lis $5
.word proc ; load address of procedure to call
jalr $5    ; call the procedure
pop($31)   ; restore callers return address
pop($29)   ; restore callers frame pointer
```

We have not yet given complete pseudo-code for how to generate code for an entire procedure. We will delay that till after discussing how a procedure call will be made. That we discuss next.

## 9.4 Arguments to a Procedure Call

Calling a procedure requires a decision on how arguments are passed to the procedure. There are two approaches. We could use registers, or we could use the stack. The code we have generated so far has used a very small subset of the general-purpose registers available on the MIPS processor. We could easily support procedures with even 20 parameters since we have that many registers still available. However, using registers to pass arguments does impose an unnecessary restriction: one can write a procedure that exceeds the number of available registers. Therefore, a fall-back mechanism is needed. The fall-back is the stack. In order to not complicate matters, we choose to not use registers at all and instead use the stack for passing all arguments.

Assuming caller-save semantics for the frame pointer, we can generate the following pseudo-code for a procedure call, i.e., the rule `factor` → `ID(`$\texttt{expr}_1, ..., \texttt{expr}_n$`)`:

```
code(factor) = push($29) + push($31) ; from above
             + code(expr_1) + push ($3)
             + code(expr_2) + push ($3)
             + ...
             + code(expr_n) + push ($3)
             + lis $5
             +.word ID   ; where ID is the name of the WLP4 procedure to call
             + jalr $5
             + pop n times ; pop all arguments
             + pop($31) + pop($29) ; from above
```

## 9.5 Generating Code for Procedures

Finally, we show how we generate code for a procedure. For the grammar rule `procedure` → `int ID(params){dcls stmts RETURN expr;}`, we have the following pseudo-code:

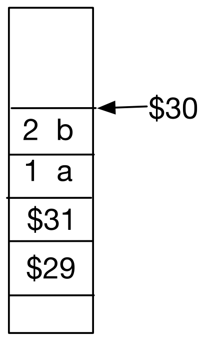
```
code(procedure) = ID:                 ; label is the name of the WLP4 procedure
          + sub $29, $30, $4         ; assuming caller-saves old frame pointer
          + push registers to save   ; assuming callee-saves registers
          + code(dcls)        ; local variables
          + code(stmts)       ; statements
          + code(expr)        ; return expression
          + pop local variables   ; since they appear above saved registers
          + pop saved registers   ; since callee-saves registers
          + jr $31
```

This code generation scheme poses an interesting dilemma. To understand the issue, let's take the example of the following WLP4 procedure and a corresponding call to the procedure, and see what the stack frame would look like when the MIPS code executes:

```
int foo(int a, int b){
   int c = 3;
   int d = 4;
   .....
}
int wain(.....){
  ... foo(1,2); ...
}
```

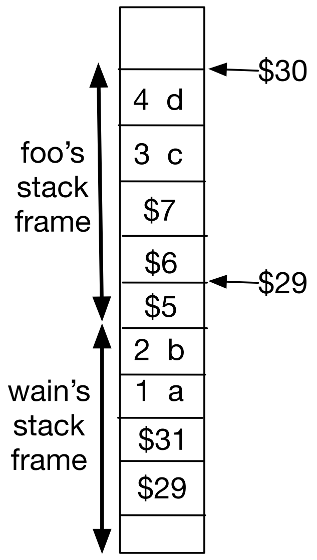
The code generated for the procedure call to foo from wain is shown to the right. Shown on the far right are the contents of the stack **at the time of execution of the jalr instruction**. We have labelled the values 1 and 2 with the parameters a and b as declared by the procedure foo.

```
push($29)
push($31)
lis $3
.word 1
push($3) ; arg1
lis $3
.word 2
push($3) ; arg2
lis $5
.word foo
jalr $5
pop ($31) ;discard arg2
pop ($31) ;discard arg1
pop($31)
pop($29)
```



Now suppose the call executes, and procedure foo begins to execute. The prologue for foo and the setup of the declarations are shown on the right. We have assumed that we are using callee-saves for registers to be preserved and that the compiler has determined that registers $5, $6 and $7 have to be preserved. On the far right, we show the contents of the stack when the procedure call has begun to execute, and the local variables have been set up.

```
foo:
sub $29, $30, $4
push($5)
push($6)
push($7)
lis $3
.word 3
push($3) ; var c
lis $3
.word 4
push($3) ; var d
```

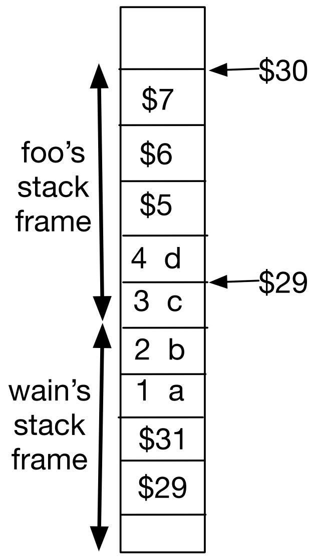


The reason for going into details was to show a problem that arises: notice the contents of the stack as seen by procedure `foo`. `foo` will need access to the parameters (`a` and `b`) and also the local variables (`c` and `d`). These are no longer contiguous in memory as we had imagined at the start of this module. Recall our discussion from Section 1 of this module: we had designed a strategy where all variables (including parameters) would be available at offsets with respect to the the frame pointer, \$29. While this is certainly still true, the values are not accessible at consistent increments of 4 starting at the offset 0. Way back when we first discussed this, we had not accounted for the fact that we will need to preserve registers. Perhaps we should have! The question to ask is whether this is a problem. The following table shows all variables available within `foo` and the offsets w.r.t. \$29.

| Symbol | Type | Offset w.r.t. \$29 |
|---|---|---|
| `a` | `int` | 8 |
| `b` | `int` | 4 |
| `c` | `int` | -12 |
| `d` | `int` | -16 |

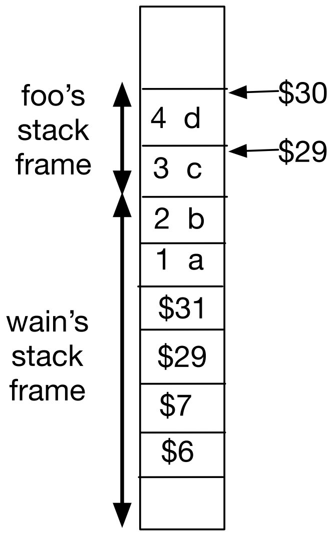
Other than the fact that this complicates where the offsets are, which could make debugging harder, the compiler can still deterministically tell where each parameter and variable are. Parameter $i$ is at $4(n-i+1)$ where $n$ is the number of parameters. This gives us the offsets 4(2-1+1)=8 for `a` which is parameter 1 and 4(2-2+1)=4 for `b` which is parameter 2. Local variable $i$ is at $-4r-4(i-1)$ where r is the number of registers to preserve. In the example above, r was 3 so local variable 1, i.e., `c` is at offset -4(3)-4(1-1)=-12 and local variable 2, i.e., `d`, is at offset -4(3)-4(2-1)=-16.

This is of course not the only way to lay out the stack. If we did want all parameters and local variables to occur contiguously in memory, one simple thing to do is switch the order in which registers are preserved, i.e., generate the code for declarations first and then push the registers that need to be preserved. If we were to make this change, the diagram on the right shows the stack contents after all declarations have been pushed on the stack and registers preserved. The offsets w.r.t. \$29 are shown below. This approach is arguably less complex for the compiler as the equation to compute offsets is straightforward. In particular, while the equation for parameters remains the same, the equation for local variable i is simply -4(i-1), i.e., number of registers that were saved no longer comes into consideration.



| Symbol | Type | Offset w.r.t. \$29 |
|---|---|---|
| `a` | `int` | 8 |
| `b` | `int` | 4 |
| `c` | `int` | 0 |
| `d` | `int` | -4 |

Of course, there is yet another possibility: the code we generated above used a **callee-save** approach to registers. We could implement **caller-save** where the caller saves all the registers that it wants to preserve. A potential configuration of the stack is shown on the right. This configuration assumes that the caller only cared about the values in $6 and $7.



## 10 A Note on Duplicate Labels

Our code generation scheme requires the generation of labels in certain situations, e.g., in `if` and `while` statements. For these, we claimed that using a counter that is incremented at each use should solve the problem. It doesn't! There is no guarantee that this would generate assembly code that is free of duplicated labels.

Can you think of a situation where despite the compiler using counters the final output contains duplicate labels?

As an example, recall that our pseudo-code for `if` uses labels `else` and `endif` concatenated with a counter. Say, we have the labels `else1` and `endif1`. There is nothing preventing the `WLP4` program from defining functions with the same name. Recall that each function gets translated to a label with the same name. So, if the input `WLP4` program had a function named `else1` (a valid if somewhat perverse name for a function), we have duplicate labels in our output assembly.

Another likely cause of such duplicates is if the `WLP4` program contains functions that have the name `init` or `print`. Recall that our generated code imports these labels. One extreme reaction could be to simply disallow the use of such function names from `WLP4` similar to how `new` and `delete` are reserved keywords. However, this is indeed extreme. An alternate approach that is recommended is that the compiler generate the labels with some thought. For example, by simply attaching a prefix to all `WLP4` function names, the problem can be eliminated. If the code generator always appends, say, `F` in front of labels corresponding to functions, and never generates a label starting with an `F` anywhere else, we guarantee that duplicate labels will not be generated. On many (but not all) Unix and Unix-like platforms, all generated function labels are prefixed with an underscore. Precisely how function labels are modified isn't important, so long as it's consistent and distinct from all other labels.

## 11 Looking ahead

We have discussed the last stage of the `WLP4` compiler. The generated assembly can be assembled and then executed on hardware that supports our variant of MIPS assembly or alternately executed in a simulator. That said, the code generated by the compiler is not ideal. Often, we chose approaches which favoured simplicity over generating efficient code. Most industrial compilers will perform compiler optimizations to generate code that is efficient. In the next module, we will briefly discuss some such optimizations.

The optimizations module is *optional*, but there are *mandatory* modules after it. While our compiler itself is complete, it depends on a runtime environment to function (remember `init`!). In the next mandatory module, we discuss one part of that runtime environment, memory management.