Warm Up Problem

What was the issue with storing the offset of variables with respect to $30?

Write **any** equivalent MIPS program corresponding to the following code:

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

Rewrite the above fixing our offset problem.

CS 241 Lecture 17

Type Checking Continued and Code Generation

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Code Generated

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

sw $1, -4($30)
sw $2, -8($30)
lis $4
.word 4
sub $30, $30, $4
sub $30, $30, $4
sw $0, -4($30) ; For int c = 0
sub $30, $30, $4     ;For int c
lw $3, 8($30)        ;8 from the symbol table
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

New Variables

Variables also have to go on the stack but we don't know what the offsets should be until we process all of the variables and parameters! For example,

```
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

As we process the code, we need to be able to compute the offset as we see the values. Also, we need to handle intermediate values of complicated arithmetic expressions by storing on the stack! We cannot do this from $30.

In Search of a Fix

How then do we arrange it so that when we see the variable, we know what the offset is?

In Search of a Fix

How then do we arrange it so that when we see the variable, we know what the offset is?

Remember that the key issue here is that $30, the top of stack frame **changes**.

In Search of a Fix

How then do we arrange it so that when we see the variable, we know what the offset is?

Remember that the key issue here is that $30, the top of stack frame **changes**. Reference the offset from the **bottom** of the stack frame! This is the value we will store in $29 (Please note that often times $30 represents this value in standard MIPS conventions).

If we calculate offsets from $29, then no matter how far we move the top of the stack, the offsets from $29 will be unchanged!

Code Generated

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

lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sw $0, -4($30)          ;For int c = 0
sub $30, $30, $4
lw $3, 0($29)           ;Offset in symbol table
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

New Variables with $29

```
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

This is easier with $29.

Something Harder

What about a more complicated program:

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

How do we handle this?

Something Harder

What about a more complicated program:

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

How do we handle this?

Well, we already have the convention that $3 stores the output so let's use this for scratch work in between.

This still isn't enough - we would need to load both `a` and `b`.

Something Harder

What about a more complicated program:

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

How do we handle this?

Well, we already have the convention that $3 stores the output so let's use this for scratch work in between.

This still isn't enough - we would need to load both `a` and `b`.

Use the convention that $5 also stores intermediate work!

More Code

```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
lw $3, 0($29)          ;Offset in symbol table (a)
add $5, $3, $0         ;Store a in $5
lw $3, -4($29)         ;Offset in symbol table (b)
sub $3, $5, $3
add $30, $30, $4       ;Optional to restore stack
add $30, $30, $4
jr $31
```

Still Have Problems

Where does this approach break down?

Still Have Problems

Where does this approach break down?

Consider adding something like $a + (b - c)$. Would need to load a, load b, load c compute $b - c$, then compute the answer. This would require a third register. (Remember, we want to process code scanning left to right).

Where should we store these values instead?

Still Have Problems

Where does this approach break down?

Consider adding something like $a + (b - c)$. Would need to load a, load b, load c compute $b - c$, then compute the answer. This would require a third register. (Remember, we want to process code scanning left to right).

Where should we store these values instead? On the stack again! This way we only will ever need two registers for scratch work!

Modify the previous code

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

lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
lw $3, 0($29)          ;Offset in symbol table (a)
sw $3, -4($30)
sub $30, $30, $4       ;Push a on stack
lw $3, -4($29)         ;Offset in symbol table (b)
add $30, $30, $4
lw $5, -4($30)          ;Pop from stack
sub $3, $5, $3
jr $31
```

Simplicity

We will use some short hands for our code. Define `code(a)` by

```
lw $3, N($29)
```

where N is the offset in the symbol table. Further, define `push($3)` by

```
sw $3, -4($30)
sub $30, $30, $4
```

and `pop($5)` by

```
add $30, $30, $4
lw $5, -4($30)
```

We are trying to find a function `code(s)` for every possible value in our grammar.

Example

Try to compute the MIPS code for

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

using these commands.

Solution

```
lis $4
.word 4
sub $29, $30, $4
push($1)
push($2)
lis $5
.word 3
push($5)
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)                    ;a <-$5
add $3, $5, $3             ;$3<- a + (b-c)
jr $31
```

Notice

- With the previous slide, we can generalize this technique so that we only need one extra register to store our computations!
- In fact, we can generalize the above to our grammar. Recall the rule

  ```
  exprA exprB PLUS term
  ```

  (where letters A and B have been added for illustrative purposes below). Then, we have

  ```
  code(exprA) =  code(exprB) +  push($3)
                + code(term) + pop($5)
                + add $3, $5, $3
  ```

  Note that above for the `add` command, `term` is in register 3 and `exprB` is in register 5.

More on Converting Grammars

Singleton grammar productions are relatively straightforward to translate:

- S $\rightarrow\vdash$ `procedure` $\dashv$ becomes `code(S) = code(procedure)`
- `expr` $\rightarrow$ `term` becomes `code(expr) = code(term)`

Assignments are also not too bad for `IDs` (pointers are a bit trickier). The production rule
`statement` $\rightarrow$ `lvalue BECOMES expr SEMI` when `lvalue` is an `ID` becomes

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

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

More on Converting Grammars

What about for `println`? Recall the rule:

statement PRINTLN LPAREN expr RPAREN SEMI

More on Converting Grammars

What about for `println`? Recall the rule:

statement PRINTLN LPAREN expr RPAREN SEMI

You actually have already done this in A2P6 and A2P7a!

Importing Code

A compiler needs to take lots of code from many different parts to make it work.

Definition

A *runtime environment* (or RTE for short) is the execution environment provided to an application or software by the operating system to assist programs in their execution. Such things include procedures, libraries, environment variables and so on.

For example, `msvcrt.dll` is a module containing standard C library functions such as `printf`, `memcpy` and so on (for Windows). For Linux machines, this is stored in `libc.so`.

Thus, it makes sense to provide `print` as part of the runtime.