Warm Up Problem

What were some of the differences between the symbol table for our assembler and our symbol table for variables?

CS 241 Lecture 16

Type Checking Continued and Code Generation

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Catching Type errors

How does one catch a type error?

- First, figure out the type of every expression using type rules (see next slides/lectures).
- If no such rule exists OR the types do not match a given rule, produce an error.
- Recall: Our ID's types are stored in the ID table.

Inference Rules For Types

Inference rules (like in CS 245!)

- If an ID is declared with type τ then it has this type:

$$\frac{\langle \text{id.name}, \tau \rangle \in \text{declarations}}{\text{id.name} : \tau}$$

- Numbers have type `int`

$$\frac{}{\text{NUM} : \texttt{int}}$$

- NULL is of type `int*`

$$\frac{}{\text{NULL} : \texttt{int*}}$$

Inference Rules For Types

Inference rules (like in CS 245!)

- Parentheses do not change the type $$\frac{E : \tau}{(E) : \tau}$$
- The Address of an `int` is of type `int*` $$\frac{E : \texttt{int}}{\&E : \texttt{int*}}$$
- Dereferencing `int*` is of type `int` $$\frac{E : \texttt{int*}}{*E : \texttt{int}}$$
- If E has type `int` then `new int[E]` is of type `int*`
$$\frac{E : \texttt{int}}{\texttt{new int}[E] : \texttt{int*}}$$

Inference Rules For Types

Arithmetic Operations

- Multiplication $$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 * E_2 : \texttt{int}}$$
- Division $$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 / E_2 : \texttt{int}}$$
- Modulo $$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 \% E_2 : \texttt{int}}$$

Inference Rules For Types

Addition and Subtraction

- Addition

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int}}$$

$$\frac{E_1 : \texttt{int*} \quad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int*}}$$

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int*}}{E_1 + E_2 : \texttt{int*}}$$

- Subtraction

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 - E_2 : \texttt{int}}$$

$$\frac{E_1 : \texttt{int*} \quad E_2 : \texttt{int}}{E_1 - E_2 : \texttt{int*}}$$

$$\frac{E_1 : \texttt{int*} \quad E_2 : \texttt{int*}}{E_1 - E_2 : \texttt{int}}$$

- Procedure Calls:

$$\frac{\langle f, \tau_1, \ldots, \tau_n \rangle \in \text{declarations} \quad E_1 : \tau_1 \quad E_2 : \tau_2 \quad \ldots \quad E_n : \tau_n}{f(E_1, \ldots, E_n) : \texttt{int}}$$

More on Types

- There is still the issue of control statements, namely:

```
while (T) {S}
if (T) {S1} else {S2}
```

- The value of `T` above should be a boolean. However WLP4 doesn't have booleans!
- Our grammar however forces `T` to be a boolean test of type `expr comp expr`.
- Hence, so long as our `exprs` are well-typed, the tests will be as well.

Inference Rules For Well-Typed

- Any expression with a type is well-typed

$$\frac{E : \tau}{\text{well-typed}(E) : \tau}$$

- Assignment is well-typed if and only if its arguments have the same type

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 = E_2)}$$

- Print is well-typed if and only if the parameter has type `int`

$$\frac{E : \texttt{int}}{\text{well-typed}(\texttt{print } E)}$$

- Deallocation is well-typed if and only if the parameter has type `int*`

$$\frac{E : \texttt{int*}}{\text{well-typed}(\texttt{delete [] } E)}$$

Inference Rules For Well-Typed

Comparisons are well-typed if and only if both arguments have the same type (either both `int` or both `int*`)

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 < E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 <= E_2)}$$

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 > E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 >= E_2)}$$

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 == E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 != E_2)}$$

Statements

- The empty sequence is well-typed

$$\frac{}{\text{well-typed}(\ )}$$

- Consecutive statements are well-typed if and only if each statement is well-typed

$$\frac{\text{well-typed}(S_1) \quad \text{well-typed}(S_2)}{\text{well-typed}(S_1; S_2)}$$

Procedures

- Procedures are well-typed if and only if the body is well-typed and the procedure returns an `int`:

$$\frac{\text{well-typed}(S) \quad E : \texttt{int}}{\text{well-typed}(\texttt{int } f(\texttt{dcl}_1, ..., \texttt{dcl}_n)\{\texttt{dcls } S \texttt{ return } E;\})}$$

- Wain is also well-typed but requires the following precise signature:

$$\frac{\texttt{dcl}_2 = \texttt{int id} \quad \text{well-typed}(S) \quad E : \texttt{int}}{\text{well-typed}(\texttt{int } f(\texttt{dcl}_1, \texttt{dcl}_2)\{\texttt{dcls } S \texttt{ return } E;\})}$$

  Notice that the first declaration can be an `int` or `int*`.

Control Statements

- All of the parts of an `if` statement are well-typed if and and only if the `if` statement itself

$$\frac{\text{well-typed}(T) \quad \text{well-typed}(S_1) \quad \text{well-typed}(S_2)}{\text{well-typed}(\texttt{if } (T)\ \{S_1\}\ \texttt{else}\ \{S_2\})}$$

- All of the parts of a `while` statement are well-typed if and and only if the `while` statement itself

$$\frac{\text{well-typed}(T) \quad \text{well-typed}(S)}{\text{well-typed}(\texttt{while } (T)\ \{S\})}$$

Assignments

There is a final sanity check with the left and right hand sides of an assignment statement.

- Given an expression, say `x=y`, notice that the left hand side and the right hand side represent different things
- The left hand side represents a place to store data; it must be a location of memory (think of variables as being memory location containers)
- The right hand side must be a value; that is, any well-typed expression.
- Anything that denotes a storage location we shall call an *lvalue*.

Example

Consider the following two snippets of code:

```
int x = 0;
x = 5;
```

This is okay; the *lvalue* `x` is a storage location

```
int x = 0;
5 = x;
```

This is **not** okay; the *lvalue* 5 is an integer and not a storage location.

For us, *lvalues* are any of variable names, dereferenced pointers and any parenthetical combinations of these. These are all forced on us by the WLP4 grammar so the checking is done for you.

Assignment Overview:

- A6: WLP4 Text File to WLP4 Tokens and lexemes (Lexical Analysis)
- A7: WLP4 Tokens and lexemes to parse tree (Syntactic Analysis)
- A8 Parse Trees to Augmented Parse Tree and Symbol Tables (Context-Sensitive Analysis)
- **A9 and 10: Augmented Parse Trees to MIPS Assembly Language (Code Generation)**

We are now on the final chapter of our journey; taking WLP4 code and converting it into MIPS assembly language.

Code Generation

- By the time we reach A9 and A10, we can assume our code has no compiler errors; that is, it is semantically and syntactically correct (no syntax, type, scoping or variable errors).
- We want to take such code and write an equivalent program in MIPS. This is our final step.
- Combined with the fact that we wrote code to take MIPS into binary; we have completed our full compiler.

Food For Thought

There are infinitely many equivalent MIPS programs to a single WLP4 program. Which should we output?

- Correctness is most important!
- For us, we seek a simplistic solution (we don't want to overcomplicate this if we can avoid it)
- Efficiency to compile (something that is exponential in the number of lines of code is likely not useful)
- Efficiency of the code itself (that is, how fast does it run?) For us, we will use number of lines as our measure of efficiency (yes this is not great but it is an easy metric!)

First Example

Let's try the following:

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

Recall our `mips.twoints` convention that registers 1 and 2 store the input values and register 3 returns the values. We need to put register 1 inside our output register $3. What MIPS command does this?

Solution

```
add $3, $1, $0
```

Almost done! What's missing?

Solution

```
add $3, $1, $0
```

Almost done! What's missing?

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

There's one hiccup...

A small concern

What is a corresponding MIPS program for this WLP4 program?

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

A small concern

What is a corresponding MIPS program for this WLP4 program?

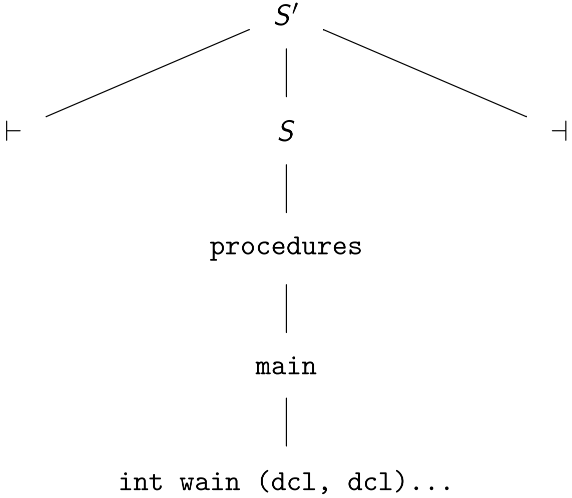
```
int wain(int a, int b){
  return b;
}
```

Answer:

```
add $3, $2, $0
jr $31
```

Seems fine but the issue is that both functions have the same parse tree!

Parse Tree



and so on.

The Main Issue

- The parse tree isn't going to be enough to determine the difference between the two pieces of code.
- How can we resolve this? How can we distinguish between these two codes?

The Main Issue

- The parse tree isn't going to be enough to determine the difference between the two pieces of code.
- How can we resolve this? How can we distinguish between these two codes?
- That's what our symbol table is for! We'll augment it to include where each symbol is stored.

Symbol	Type	Location
`a`	`int`	$1
`b`	`int`	$2

The Main Issue

- The parse tree isn't going to be enough to determine the difference between the two pieces of code.
- How can we resolve this? How can we distinguish between these two codes?
- That's what our symbol table is for! We'll augment it to include where each symbol is stored.

Symbol	Type	Location
`a`	`int`	$1
`b`	`int`	$2

... by now however you know nothing is ever this easy. What can go wrong?

Issues

- Storing variables and parameters in registers will force us to run out of registers quickly (Think about recursive code!)
- This means we need to store them somewhere else? Where?

Issues

- Storing variables and parameters in registers will force us to run out of registers quickly (Think about recursive code!)
- This means we need to store them somewhere else? Where?
- Store them on the stack!

Code Generation

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

Becomes

```
sw $1, -4($30)
sw $2, -8($30)
lis $4
.word 4
sub $30, $30, $4
sub $30, $30, $4
lw $3, 4($30)
add $30, $30, $4
add $30, $30, $4
jr $31
```

We make the convention that $4 always contains the number 4.

Updates to Symbol Table

Instead of the symbol table storing registers, it should actually store the offset from the stack pointer!

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

Unfortunately, this also gives us problems - why?

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;}
```

Let's generate the code for this program (see next slide)

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
sub $30, $30, $4     ;For int c
sw $0, 0($30)        ; c = 0
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
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!

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)
```

Try to compute the code for $a + (b - c)$ using these commands.