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 15

Type Checking Continued and Code Generation
With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Types in WLP4

- In WLP4, there are two types: `int` and `int*` for integers and pointers to integers.
  - (This restriction is based on C's predecessor, B!)
- For type checking, we need to evaluate the types of expressions and then ensure that the operations we use between types corresponds correctly.

Types in WLP4

- If given a variable in the wild, how do we determine its type?
- Use its declaration! Need to add this to the symbol table.

Symbol Table Implementation

We can use a global variable to keep track of the symbol table:

```
map<string, string> symbolTable; // name -> type
```


but by now you know nothing is ever this easy! What can go wrong?

- This doesn’t take scoping into account!
- Also need something for functions/declarations!

Issues

- Consider the following code (specifically with x). Is there an error?

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int wain(int x, int y) {
    return foo(y) + x;
}
```

- No! Duplicated variables in different procedures are okay!

Issues

- Is the following an error?

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int wain(int a, int b) {
    return foo(b) + x;
}
```

- Yes! The variable *x* is not in scope in *wain*!

Issues

- Is the following an error?

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int foo(int b) { return b; }
int wain(int a, int b) {
    return foo(b) + a;
}
```

- Yes! We have multiple declarations of *foo*.

Resolution

- How shall we resolve this?
- Probably want a separate symbol table per procedure…
- *Also* need a global symbol table!
- In larger compilers, need a *tree of symbol tables* that mirrors the parse tree!
- See notes for code suggestions.

Not Quite Enough

- A symbol table is a map… what are we mapping each symbol to?
- For type checking, we need to know the type of each symbol.
  - In WLP4, a string could be sufficient, but you should use a data structure so you can add more later (codegen will want more!).
- What about procedures?

Procedure Signature

- Procedures don't have types, they have *signatures*[1]
  - The signature of a procedure is the types of its inputs (arguments) and output (return)
- In WLP4, all procedures return `int`, so we really just need the argument types: an array of types

[1] In languages with first-class functions, the signature is the type, but WLP4 doesn't have first-class functions, so they're distinct

Computing Signature

- Simply need to traverse nodes in your parse tree of these forms:
  - params ->
  - params -> paramlist
  - paramlist -> dcl
  - paramlist -> dcl
  - paramlist -> dcl COMMA paramlist
- Again, all of this can be done in a single pass.

An Example

- Consider

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int wain(int *a, int b) {
    return foo(b) + a;
}
```

- Global symbol table:
  - foo: [int], wain: [int*, int]
- Local symbol tables:
  - foo: a: int, x: int
  - wain: a: int *, b: int

wain is special in WLP4. You probably don't need it in your symbol table!

Type Errors

- What are type errors and how to find them?
- Two separate issues:
  - What are type errors? (Definition)
  - How to find them? (Implementation)

Definition of Type (Errors)

- Need a set of rules to tell us:
  - The type of every expression
  - Whether an expression makes sense with the types of its subexpressions
  - Whether a statement makes sense with the types of its subexpressions

Detection of Type (Errors)

- There’s really only one algorithm with a tree: traverse the tree!
- Implement a (mostly) post-order traversal that applies defined rules based on which expressions it encounters

Inference Rules For Types

Inference rules are Post rules (like in CS 245!)

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

$$\frac{(\text{id.name}, \tau) \in \text{declarations}}{\text{id.name} : \tau}$$

- Numbers have type `int` $\dfrac{}{\text{NUM} : \texttt{int}}$

- NULL is of type `int*` $\dfrac{}{\text{NULL} : \texttt{int*}}$

Inference Rules for Types

- Inference rules are the “true” case. If no inference rule matches, that means the expression or statement doesn’t type check: type error!
- Look for good, not for bad: errors should always be the “else” case
- Note: We’re about to go over a bunch of rules, but you should refer to the notes for them, not here

Type Errors

- What are type errors and how to find them?
- Two separate issues:
  - What are type errors? (Definition)
  - How to find them? (Implementation)

Definition of Type (Errors)

- Need a set of rules to tell us:
  - The type of every expression
  - Whether an expression makes sense with the types of its subexpressions
  - Whether a statement makes sense with the types of its subexpressions

Detection of Type (Errors)

- There’s really only one algorithm with a tree: traverse the tree!
- Implement a (mostly) post-order traversal that applies defined rules based on which expressions it encounters

Inference Rules For Types

Inference rules are Post rules (like in CS 245!)

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

$$\frac{(\text{id.name}, \tau) \in \text{declarations}}{\text{id.name} : \tau}$$

- Numbers have type `int` $\qquad \overline{\text{NUM} : \texttt{int}}$

- NULL is of type `int*` $\qquad \overline{\text{NULL} : \texttt{int*}}$

Inference Rules for Types

- Inference rules are the "true" case. If no inference rule matches, that means the expression or statement doesn't type check: type error!
- Look for good, not for bad: errors should always be the "else" case
- Note: We're about to go over a bunch of rules, but you should refer to the notes for them, not here

Inference Rules For Types

- 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

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

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

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

- 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:

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

More on Types

- The basic kind of statement type is an *expression statement*. An expression statement is OK as long as the expression has a type (any type!)
- There is still the issue of control statements.
- Statements don't have a type, but can be "well typed".

More on Types

- There is still the issue of control statements, namely:
  - while (T) { S }
  - if (T) { S_1 } else { S_2 }
- The value of T above should be a boolean. But WLP4 doesn't have booleans!
- Our grammar forces it to be a boolean expression, so we don't need to check that.
- But, we still need to check its subexpressions!

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*`)

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

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

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

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`:

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

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

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

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

Control Statements

- An `if` statement is well-typed if and and only if all of its components are well-typed

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

- A `while` statement is well-typed if and and only if all of its components are well-typed

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

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 is 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.

Type-Checking Recommendations

- Brush up on recursion; almost everything from this point on is traversing a tree.
- Remember that C is your comparison. If you're not sure what the right option is, ask what C does.
- WRITE TINY TEST CASES. Don't count on just our test cases!

Example

- Let’s type-check a tree. We’ll use the code from last time that we used to demonstrate a symbol table

Assignment Overview:

- A4: WLP4 Text File to WLP4 Tokens and lexemes (Lexical Analysis)
- A5: WLP4 Tokens and lexemes to parse tree (Syntactic Analysis)
- A6: Parse Trees to Typed Intermediate file (and symbol tables) (Context-Sensitive Analysis)
- **A7 and 8: Typed Intermediate File 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.

(There are still things to do after this, but that's the epilogue ☺)

Food For Thought

There are infinitely many equivalent MIPS programs for 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?)