Tutorial 9

- Name & Type Error Checking in WLP4

Types of Errors

- **Name errors**: errors related to identifiers & their meaning(s)
  - Eg: name used but not defined
  - Eg: name used multiple times in different contexts with no way to disambiguate

- **Type errors**: errors related to the type of the expression
  - I.e: determining correct type usage (int + int) vs. incorrect type usage (int* + int*)

We can detect semantic errors by traversing & analyzing the parse trees of our expressions

- Idea: traverse our parse tree to gather info on the prog so we can enforce rules that need context sensitive information

Name Errors

- 2 types of identifiers in WLP4: variable & procedure names.
- Types of name errors in WLP4:
  1. Duplicate declarations
  2. Use without declaration

Complications also arise due to WLP4 supporting scope
eg 1)
\[
\text{int } f() \equiv
\begin{align*}
\text{int } x &= 0; \\
\text{return } x;
\end{align*}
\]
\[
\text{int } f() \equiv
\begin{align*}
\text{int } x &= 0; \\
\text{return } x;
\end{align*}
\]
\[
\text{int } wain(\text{int } a, \text{int } b) \equiv
\begin{align*}
\text{int } x &= 0; \\
\text{return } x;
\end{align*}
\]
\[
\text{int } wain(\text{int } a, \text{int } b) \equiv
\begin{align*}
\text{int } x &= 0; \\
\text{return } x;
\end{align*}
\]
Valid: \(x\) defined \(\geq 1\)
Invalid: \(x\) used in \(wain\) & not
times, but in local scopes defined in \(wain\)

eg 2)
\[
\text{int } f() \equiv
\begin{align*}
\text{int } f &= 1; \\
\text{return } f;
\end{align*}
\]
\[
\text{int } f() \equiv
\begin{align*}
\text{int } x &= 0; \\
\text{return } x;
\end{align*}
\]
\[
\text{int } g(\text{int } g) \equiv
\begin{align*}
\text{return } g-1;
\end{align*}
\]
\[
\text{int } f() \equiv
\begin{align*}
\text{int } x &= 0; \\
\text{return } x;
\end{align*}
\]
\[
\text{int } wain(\text{int } a, \text{int } b) \equiv
\begin{align*}
\text{return } g(a) + f() + a
\end{align*}
\]
\[
\text{int } wain(\text{int } x, \text{int } y) \equiv
\begin{align*}
\text{return } f() + x;
\end{align*}
\]
Valid: Ability to distinguish between the \(fns\) & \(vars\) in all scopes!
Invalid: Cannot have multiple declarations in the same scope
Problem: We need context to determine if vars/fns are valid

Idea: Use a symbol table! (like we did for MIPS labels!)

L> When you encounter a declaration, add it to the table.
   → If it is already there, error
   → lets you distinguish scope & types of vars/fns

L> When you encounter a use, check if it is in the table. If not, error

L> WIPLH enforces forward declaration → only need 1 pass!

Procedure rules:

procedure → INT ID LPAREN params RPAREN
             LBRACE dcls statements RETURN expr SEMI RBRACE
main → INT WAIN LPAREN dcl COMMA dcl RPAREN
       LBRACE dcls statements RETURN expr SEMI RBRACE

* Since 'params'/dcls' occur before 'statements', our code will
  have made a complete symbol table before running any code!*

* We can check our "declaration before use" rule on variables
  by traversing the 'statements' subtree
  L> traverse to all leaf nodes from the rules:
    Factor → ID
    Ivalue → ID
  L> check if ID is in your symbol table, if not then error!
For multiple scopes from adding procedures

- Create a global symbol table \forall procedures
- All procedures should have their own local symbol tables
- Global table maps:
  - procedure ID to \langle \text{procedure signature}, \text{local symbol table} \rangle

```
procedure \rightarrow \text{INT } \text{ID LPAREN } \text{params RPAREN}
LBRACE \text{dcls statements RETURN expr SEMI RBRACE}
main \rightarrow \text{INT WAIN LPAREN dcl COMMA dcl RPAREN}
LBRACE dcls statements RETURN expr SEMI RBRACE
```

With this, we can explore the `Statements`/`expr` subtrees to find var uses & in calls to check in the global symbol table
eg 1 symbol tables:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Signature</th>
<th>Var</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>[]</td>
<td>x</td>
<td>int</td>
</tr>
<tr>
<td>wain</td>
<td>[int,int]</td>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td>int</td>
</tr>
</tbody>
</table>

```
int f() {
    int x = 0;
    return x;
}
```

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

2 different locally defined x's!

```
int f() {
    int x = 0;
    return x;
}
```

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

No x in our symbol table, error!
eg. 2 symbol tables

\[
\begin{align*}
\text{int } f(x) & : \\
\text{int } f = 1; & \\
\text{return } f; & \\
\hline
\text{Symbol table} & \\
\text{Procedure} & \text{Signature} & \text{Var} & \text{type} & \\
\hline
f & [] & f & \text{int} & \\
& & & & \\
\text{int } g(x) & : \\
\text{return } g - 1; & \\
\hline
\text{Symbol table} & \\
\text{Procedure} & \text{Signature} & \text{Var} & \text{type} & \\
\hline
\text{int } wain(x, y) & : \\
\text{return } g(x) + f(y) + a & \\
\hline
\text{int } f(x) & : \\
\text{int } x = 0; & \\
\text{return } x; & \\
\hline
\text{Symbol table} & \\
\text{Procedure} & \text{Signature} & \text{Var} & \text{type} & \\
\hline
f & [] & x & \text{int} & \\
& & & & \\
\text{int } f(x) & : \\
\text{int } x = 0; & \\
\text{return } x; & \\
\hline
\text{Symbol table} & \\
\text{Procedure} & \text{Signature} & \text{Var} & \text{type} & \\
\hline
\text{int } wain(x, y) & : \\
\text{return } f(x) + x; & \\
\hline
\text{Note: this error would have been thrown when we 1st reached a duplicate } f(x) \text{ in symbol table creation!}
\end{align*}
\]
Type Errors

For all nodes in a parse tree, cache their types.

Traverse our parse tree by recursing on children then parent:

- For leaf nodes, get & cache their type from the global symbol table.
- For non-leaf nodes, assume children have their type cached & use them to compute their type.

Types from expr's come from course reference sheet:

- `int` PLUS `int` → `int`    ? And many more!
- `int*` PLUS `int` → `int*`  \( \} \) Help us determine all node types!
- `int` PLUS `int*` → `error` \( \)|

* An assignment (ie: lvalue BECOMES expr SEMI) is well-typed if the lvalue & expr have the same type.
  → else, error

**eg:** Given the following WLP4 code:

```wlp4
int foo(int x, int y) {
    return x + 7 * y + 1;
}
```

```wlp4
...  

int a = 0;
int b = 0;
int* c = NULL;
int* d = NULL;
```

Determine if the following WLP4 code is well-typed.
1) \((d + (((c-d*b)+d)-(c+(a*b)))) = \text{lv} \text{value (LHS)}\)
\((c-d+*\text{new int}[d+b-c])i \text{expr (RHS)}\)

**Goal:** Show the type of LHS = type of RHS
(Otherwise the statement is not well typed)

- Parse tree:

  ![Diagam](image)

  Break it down to see if LHS type = RHS type
Starting with our `value: ( token : type )` *(d + ((c - &b) + d) - (c + (a * b)))*

```
value: ?
  STAR
    factor: ?
  LPAREN
    expr: ?
    RPAREN
  expr: ?
    PLUS
    term: ?
  term: ?
    factor: ?
  factor: ?
    LPAREN
    expr: ?
    RPAREN
  ID d: ?
  expr: ?
    MINUS
    term: ?
    term: ?
    factor: ?
  factor: ?
    LPAREN
    expr: ?
    RPAREN
    expr: ?
    PLUS
    term: ?
    term: ?
    factor: ?
    factor: ?
    LPAREN
    expr
    RPAREN
    ID d: ?
    ID c: ?
    term: ?
  LPAREN
  expr: ?
  RPAREN
  expr: ?
  PLUS
  term: ?
  term: ?
  factor: ?
  factor: ?
  LPAREN
  expr
  RPAREN
  ID d: ?
  ID c: ?
  term: ?
  STAR
  factor: ?
  expr: ?
  MINUS
  term: ?
  term: ?
  factor: ?
  ID b: ?
  ID a: ?
  factor: ?
  AMP
  value: ?
```

Step 1) Fill in types for all ID's we see from our symbol table (i.e. types from the provided code)
Step 2) Propagate upwards through parents with single children
Step 3) From rule conditions in our reference sheet, get updated type & repeat steps.

1. Continue until your root gets a type! This is the type of the entire expression!
2. If rule condition with a given type is not listed or met, error & exit!
For the RHS:

\[(c - d + * new \ int[d + b - c])\]

expr: ?
term: ?
factor: ?

\[\text{LPAREN} \text{ expr: ? } \text{RPAREN}\]

expr: ? \ PLUS \ term: ?

expr: ? \ MINUS \ term: ?

factor: ? \ STAR \ factor: ?

factor: ? \ ID d: ? \ NEW \ INT \ LBRACK \ expr: ? \ RBRACK

ID c: ?

expr: ? \ MINUS \ term: ?

expr: ? \ PLUS \ term: ?

term: ? \ factor: ? \ ID c: ?

factor: ? \ ID b: ?

ID d: ?
\[ \text{expr: int} \leftarrow \text{well-typed & RHS type = int} \]

\[ \text{term: int} \]

\[ \text{factor: int} \]

\[ \text{LPAREN expr: int RPAREN} \]

\[ \text{expr: int PLUS term: int} \]

\[ \text{expr: int MINUS term: int} \]

\[ \text{factor: int STAR factor: int} \]

\[ \text{factor: int PLUS factor: int} \]

\[ \text{factor: int STAR factor: int} \]

\[ \text{factor: int PLUS factor: int} \]

\[ \text{ID d: int NEW INT LBRACK expr: int RBRACK} \]

\[ \text{ID c: int} \]

\[ \text{ID a: int} \]

\[ \text{ID b: int} \]

\[ \text{ID d: int} \]

\[ \text{Thus:} \]

\[ \text{Statement type: int} \leftarrow \text{.. 1 is well-typed!} \]

\[ \text{Value type: int} \]

\[ \text{BECOMES} \]

\[ \text{expr type: int} \]

\[ \text{Satisfies assignment requirement of LHS = RHS} \]
if \((*(c+a\%b) < (da-db))\) 3

\[
\text{println}(\text{\#\#}d*\text{\#\#}c-(\text{\#\#}b));
\]

3 else 3

\[
delete [] d+da-c;
\]

3

• Break it apart, starting from the inner `if`:

\[
(\text{Note: this is a quicker way to perform type checking})
\]

\[
*(c+a\%b) < (da-db)
\]

\[
\begin{align*}
  & int * \% int \\
  \downarrow & int \\
  int * & int
\end{align*}
\]

\[
int < int
\]

\[
int <- \text{if body is well-typed}
\]

• Now the `if` body:

\[
\text{println}(\text{\#\#}d*\text{\#\#}c-(\text{\#\#}b));
\]

\[
\begin{align*}
  & int * \\
  \downarrow & int \\
  int * & int
\end{align*}
\]

\[
\text{PRINTLN }<\text{valid, PRINTLN's expr needs to be type } int
\]

\[
\text{thus println is well-typed.}
\]
The else body:

```c
delete [] *d + &a - c
```

```
*int* &int
↓  ↓
int + int*
↓  ←
int* - int*
```

DELETE [] int ← INVALID! delete [] requires an int*

```
• 2) is not well-typed since delete [] *d + &a - c
```

is not well-typed