CS 473: COMPILER DESIGN
Abstract Syntax Trees

Source Code (Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:
If
  Eq
  Assn
  None
  b
  0
  a
  1

Assembly Code
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
...
Lexing and parsing deal with questions of syntax:
- What symbols are allowed to appear in programs? What are the pieces of programs? What are the fundamental operations and structures?

The rest of the compiler looks at the semantics of programs:
- What does this operator/command do? What is the output of this supposed to be? How do we translate into something that does the same thing?

We start by gathering information from the program structure, and looking for errors in that information: semantic analysis

<table>
<thead>
<tr>
<th>Syntax errors:</th>
<th>Semantic errors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int b 4;</td>
<td>int b = “hi”;</td>
</tr>
<tr>
<td>y x &lt;;</td>
<td>a = b + 1;</td>
</tr>
<tr>
<td>if w x y z;</td>
<td></td>
</tr>
</tbody>
</table>
AST Analysis

• From now on, a program is just a data structure (usually some kind of tree/graph)

• So we can answer questions about the program by writing functions on data structures!

• Trees are fundamentally recursive data structures, so we’ll usually analyze them with recursive functions

• demo: analysis.c
• Trees are fundamentally recursive data structures, so we’ll usually analyze them with recursive functions

• One function per nonterminal

print_numbers_exp(exp_node* e){ … }
print_numbers_stmt(stmt_node* s){ … }

• Switch on type of node, then compute the value, with recursive calls for any relevant children

if(prog == NULL) return;
switch(prog->kind){
  case assign_stmt:
    print_numbers_exp(prog->data.assign_ops.rhs); return;
  case if_stmt:
    print_numbers_exp(prog->data.if_ops.cond);
    print_numbers(prog->data.if_ops.then_stmt);
    print_numbers(prog->data.if_ops.else_stmt);
}
Questions

Top
Type Checking

• Goal: catch programs that we know won’t run properly, before we bother translating them

```c
int a = "Hi";
```

• Avoid wasting time and effort translating something that won’t mean anything
• Avoid translating into machine code that does mean something, but the wrong thing

```c
a = 0x123456
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x123456</td>
<td>H</td>
</tr>
<tr>
<td>0x123457</td>
<td>i</td>
</tr>
</tbody>
</table>

• Approach: recurse through the AST, remembering type information when we see it, giving errors when we find type mismatches
• For each kind of AST node, we need to know when it’s type-correct and what type it has (if any)
Exercise: What types exist in your favorite programming language?

Basic types: int, string

123 "Hi!" int a = 3;

Arrays

int[] arr1 = 3; // not valid C syntax, but it’s allowed in our language!

y = arr1[2];

Other languages also have struct/record types, object types, void type, function types, ...
Type Checking: Expressions

• Expressions evaluate to values, so every expression has a type
• Expressions may also do some computation, so they may have some conditions for correctness (for instance, + can only add numbers)

• Kinds of expressions: literals, variables, operations, function calls, ...
  – For each one, we have to figure out what type it produces, and maybe also check to make sure it’s well-typed

• In code:
  
  ```c
  ty_node* type_of_exp(exp_node* e){
    switch(e->kind){
      case int_exp:
      case string_exp:
        ...
    }
  }
  ```

• Literals: have the associated basic type, never have a type error

1

   type: int

"Hi"

   type: string
Type Checking: Expressions

- Expressions evaluate to values, so every expression has a type
- Expressions may also do some computation, so they may have some conditions for correctness (for instance, + can only add numbers)

- Kinds of expressions: literals, variables, operations, function calls, ...
  - For each one, we have to figure out what type it produces, and maybe also check to make sure it’s well-typed

- In code:
  ty_node* type_of_exp(exp_node* e){
    switch(e->kind){
      case int_exp: return IntTyNode();
      case string_exp: return StringTyNode();
      ... }

- Literals: have the associated basic type, never have a type error

<table>
<thead>
<tr>
<th>1</th>
<th>&quot;Hi&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: int</td>
<td>type: string</td>
</tr>
</tbody>
</table>
Type Checking: Expressions

- Expressions evaluate to values, so every expression has a type.
- Expressions may also do some computation, so they may have some conditions for correctness (for instance, + can only add numbers).

- Operations: check types of operands, return type of result (each operator has its own expected input and output types).

```
Plus
1
2
```

```
Plus
1
"a"
```

Type: int
Type: int
Type: int
Type: string

Type error!
Type Checking: Expressions

- Expressions evaluate to values, so every expression has a type.
- Expressions may also do some computation, so they may have some conditions for correctness (for instance, + can only add numbers).

- Operations: check types of operands, return type of result (each operator has its own expected input and output types).
- “Check types of operands” is recursive: operands can themselves be complex expressions!
Program 3: Type Checker

• Posted on the course website (https://www.cs.uic.edu/~mansky/teaching/cs473/sp21/program3.html)
• Fill in pieces of a type checker
• Due next Monday at the start of class
• Submit via Gradescope

• Writing an analysis function on ASTs:
  1. One function for each nonterminal (expressions, statements, etc.)
  2. In each function, a switch statement over the kinds of nodes for that nonterminal
  3. In each case, perform the check/compute the value for that case, and make recursive calls to any child nodes
We need to know a’s type from somewhere else! (usually a variable declaration)
We need to know a’s type from somewhere else! (usually a variable declaration)
Types for Variables: Environments

• In most programming languages, the main operation is assigning values to variables
  – In most languages, we need to know the types of variables
  – In most languages, we also need to know what variables are in scope

• We store this information in environments (a.k.a. symbol tables)

```c
int a = 3;
string b = “Hi”;
```

```
{a : int, b : string}
```

“binding for b”

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>string</td>
</tr>
</tbody>
</table>
In most programming languages, the main operation is assigning values to variables.
- In most languages, we need to know the types of variables.
- In most languages, we also need to know what variables are in scope.

We store this information in environments (a.k.a., symbol tables).

```
int a = 3;
string b = "Hi";
{a : int, b : string}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>string</td>
</tr>
</tbody>
</table>
In most programming languages, the main operation is assigning values to variables.

- In most languages, we need to know the types of variables.
- In most languages, we also need to know what variables are in scope.

We store this information in environments (alternatively, symbol tables).

```plaintext
int a;
string b;

fun f return int(int b){
   string a = “a”;
   print(a);
}
```
Scope and Types

• In most programming languages, the main operation is assigning values to variables
  – In most languages, we need to know the types of variables
  – In most languages, we also need to know what variables are in scope

• We store this information in environments (alternatively, symbol tables)

```plaintext
int a;
string b; // { a : int, b : string }

fun f returns int(int b){ // { a : int, b : int }
    string a = "a"; // { a : string, b : int }
    print(a);
}

// { a : int, b : string }
```
We get the type of a variable from the environment
We get the type of a variable from the environment

- Type: string

Type Checking: Variables

1 + a

Vars: {a : string, ...}

- type error!
We get the type of a variable from the environment.

- We get the type of a variable from the environment.
Questions

Top
Array access acts like an operator, taking a T[] and an int and returning a T
Our environments should also record function signatures (argument and return types)
Type Checking: Functions

f(a, 1)

Our environments should also record function signatures (argument and return types)

Look up function signature in the environment, check arguments, result is return type

• Our environments should also record function signatures (argument and return types)
• Look up function signature in the environment, check arguments, result is return type
Questions

Top
In most languages, statements don’t return values, so they don’t have types.
But they still use values, so we need to check that they’re type-correct.
And they also might add information to the type environment.
Kinds of statements: assignment, declarations, control flow, ...
  – For each one, we have to check to make sure it’s well-typed, and add any newly declared items to the environment.

```
Assign
{ a : int, ... }
```

```
OK
```

```
a type: int
5 type: int
```
• In most languages, statements don’t return values, so they don’t have types
• But they still use values, so we need to check that they’re type-correct
• And they also might add information to the type environment
• Kinds of statements: assignment, declarations, control flow, ...
  – For each one, we have to check to make sure it’s well-typed, and add any newly declared items to the environment

```
Assign
  a
  “hi”
{ a : string, ... }
```

OK
Type Checking: Statements

• In most languages, statements don’t return values, so they don’t have types
• But they still use values, so we need to check that they’re type-correct
• And they also might add information to the type environment
• Kinds of statements: assignment, declarations, control flow, ...
  – For each one, we have to check to make sure it’s well-typed, and add any newly declared items to the environment

• We can assign a variable any value, but it has to match the variable’s type
• Exercise: fill in typechecking code for assign statements in typecheck.c
Type Checking: Declarations

• The bindings in environments come from *declarations*
• Variable declarations:

```
Vardec {a : int}

{a : int}
Vardec

int  a

{a : int}
Vardec

int  a  5

```

• For a declaration-and-definition, we need to make sure the value matches the declared type!
Function declarations:

```plaintext
fun f returns int (int x, int y) {
    return x + y;
}
```

First, typecheck the body
Function declarations:

fun f returns int (int x, int y) {
    return x + y;
}

First, typecheck the body
   - in an environment that includes the parameters!

Then, add the function signature to the top-level environment
Questions

Top
Recursive function declarations:

\[
\text{fun } f \text{ returns } \text{int} \ (\text{int } x, \text{ int } y) \ {\{} \\
\quad \text{if}(y == 0) \text{ return } 1; \\
\quad \text{else return } x + f(y - 1, x); \\
\{ \text{x : int, y : int} \} \\
\]

• First, typecheck the body
  – in an environment that includes the parameters!
Type Checking: Recursive Declarations

• Recursive function declarations:

```plaintext
def f(int x, int y):
    if(y == 0):
        return 1;
    else:
        return x + f(y - 1, x);
```

• First, typecheck the body
  – in an environment that includes the parameters, and the function signature!
  – We take the function for granted while type-checking the function itself!

• Then, add the function signature to the top-level environment
Questions

Top
Semantic Analysis: Summary

• Goal: gather information about elements of the program, and catch any errors that were too tricky for parsing to catch (type errors, scope errors)

• Method: recurse through the AST, building up information (stored in environments/symbol tables) and checking for errors

• Result: the same AST as before, but now we know it’s free from common errors and worth compiling!

• Type checking can get much more complicated, with generics, classes, first-class function types, ...
  — See also CS 476: Programming Language Design
Questions

Top
Feedback Poll

• A short anonymous poll to see how things are going
• Link: https://forms.gle/96pMP8ehz9Et9vDQ7

• Once you’ve done it, just submit the word “completed” for Exercise 2/19 on Gradescope

• This will help me make the course better, so thank you!