Catch errors that are not caught by parsing (because many constructs are not context-free).

Examples (C):

1. All identifiers are declared
2. Types
3. ...

The set of legal programs is a subset of the parse-able programs.

Scope

- Match identifier declarations with uses
    - Important static analysis step

  Example 1:

```
        void foo (int n) {
          int a = 0, i;
    printf("a = %d\n", a);
            for (i = 0; i < n; i++) {
      int a = 1;
      printf("a + j = %d\n", a + j); //a's value is 1. where is j. error?
            }
    printf("a = %d\n", a);
  }
```

- The *scope* of an identifier is the porition of the program in which that identifier is accessible
- The same identifier may refer to different things in different parts of the program
    - Different scopes for same name don't overlap
- An identifier may have restricted scope
- Most languages have *static* scope
    - Scope depends only on the program text, not run-time behaviour. e.g., C
- A few languages are *dynamically* scoped
    - Lisp (earlier versions, now moved to static scoping), SNOBOL
    - Scope depends on execution behaviour of a program
- A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program
    - Example:

```
void foo(int x)
{
  int a = 4;
  bar(3);
}
void bar(int y)
{
  printf("a = %d\n", a); //refers to a in the closest enclosing binding in the execution of the program
}
```

- We will talk about dynamic scope later ...
- In C, identifier bindings are introduced by:
    - Function definitions (introduces method names; only allowed at top-level)
    - Struct definitions (introduces type names aka class names)
    - Variable declarations (introduces objects of certain types; objects represent regions of memory)
    - Structure field definitions (introduces objects)
    - Function argument declarations (introduces objects of certain types)
    - Typedefs (introduces type names)
- Not all identifiers follow the most-closely nested rule
    - Function definitions in C cannot be nested
    - Forward declarations are allowed for functions and variables (extern keyword). Thus a variable can be used before it is defined.
    - In C++, you can use member functions for the type-variables even if they have never been declared/defined (type-variables are the arguments to the template keyword)
    - In C++ class declaration, you can use a member variable in a method before it is defined.
    - Some languages allow identifiers to be overridden within the same scope, others don't. Some languages allow identifiers to be overridden in different nested scopes, others may not.

Symbol Tables

- Much of semantic analysis can be expressed as a recursive descent of an AST
    - *Before*: Process an AST node `n`
    - *Recurse*: Process the children of `n`
    - *After*: Finish processing the AST node `n`
- When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined
- A *symbol table* is a data structure that tracks the current bindings of the identifiers
- Example: the scope of a variable declaration is one subtree of the AST

```
{ //new scope
  int x = 4;  //declarations
  f(x);
```

```
    }
    Tree:
    NewScope -->(left child) declarations x = 4
    NewScope -->(right child) statements
```

On entry to the new scope, the new declarations will be added to the symbol table before recursing down right child (statements). On returning from the new scope, the new declarations will be removed and the old declarations of x (if any) will be restored in the symbol table.

- For implementing a symbol table, we can use a stack of scopes (assuming C89)
    - `enter_scope()`: start a new nested scope
    - `find_symbol(x)`: search stack starting from top, for the first scope that contains `x`. Return first `x` found or NULL if none found
    - `add_symbol(x)`: add a symbol to the current scope (top scope)
    - `check_scope(x)`: true if `x` defined in the current scope (top scope). allows to check for double definitions.
    - `exit_scope()`: exit current scope
- For implementing forward declarations, or C++ style use-before-define styles, one may have to make multiple passes over the AST. First pass: gather all member names. Second pass: Do the checking. In general, semantic analysis requires multiple passes (often more than two). Easier to break the analysis into simpler passes, as opposed to one big complicated pass.

Types

- What is a type
    - The notion varies from language to language
    - Typically
        - A set of values
        - A set of operations on those values
    - Classes are one instantiation of the modern notion of type
- Consider the assembly language fragment:

  ```
  add $r1,$r2,$r3
  ```

  What are the types of $r1, $r2, $r3? Can't say. For all you know, r1 may represent a bit-pattern that represents a string. The only thing we can say is that these have the base-type *bitvector*.
- Types restrict the set of legal programs further
    - e.g., addition on a string seems quite unlikely
    - Tradeoff: catch common mistakes (e.g., addition to string is quite unlikely), but may eliminate some more-optimized programs (we can still specify all programming functionality, just that the remaining set of programs may not be as optimized as the original set of programs).
    - Different programming languages take different tradeoff points, e.g., OCaml has immutable variables (i.e., values are written only at time of creation) while C has mutable variables (values can be written at any time).
    - Well-typed programs (or well-defined programs) are a subset of parseable programs that have all symbol uses matched with its corresponding definition.
        - A type checking logic can execute either at compile time or at runtime to disambiguate well-typed programs from not-well-typed programs
        - *or* this checking responsibility can be left to the programmer (undefined behaviour)
- Certain operations are legal for values of each type
    - It doesn't make sense to add a function pointer and an integer in C.
    - It does make sense to add two integers
    - But both have the same assembly language implementation!
- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with correct types
    - Enforces intended interpretation of values, because nothing else will!
- Three kind of languages
    - Statically typed: All or almost all checking of types is done as part of compilation (C, Java)
    - Dynamically typed: Almost all checking of types is done as part of program execution (Scheme, Lisp, Python, Perl)
    - Untyped: No type checking. All strings in the language (e.g., CFG) are valid strings. e.g., machine code
- Another variation of typing: undefined behaviour. Accessing word beyond the length of an array is not type-checked in C. But if this happens at runtime, the program is not well-defined (or well-typed). The type-system is in programmer's head, but nobody will check it for you!
- Competing views on static vs. dynamic typing
    - Long-standing debate which is better
- Static typing proponents say:
    - Static checking catches many programming errors at compile time
    - Avoids overhead of runtime checks
- Dynamic typing proponents say:
    - Static type systems are restrictive. Restricts the programs that you can write (even though they may be well-typed at runtime)
        - Array de-reference quite hard to verify statically. Restricting programs to only those programs that can be verified statically is a non-starter. Java decides to use runtime checks (dynamic type-checking).
    - Rapid prototyping difficult within a static type system
- Some strange things about typing
    - A lot of code is written in statically typed languages with an "escape" mechanism
        - Unsafe casts in C, Java
        - Philosophy: if the programmer knows what she is doing, let her do whatever she wants. e.g., addition on strings. If the programmer is making a mistake, the program will behave badly, but the language does not provide any guarantees (after all you took the matter in your hands)
    - People retrofit static typing to dynamically typed languages

        - Avoid runtime costs
        - Debugging
        - This can only be best-effort, no guarantees. Some dynamic checks may remain
- In C
    - The user declares types for identifiers
    - The compiler infers types for expressions
- *Type checking* is the process of verifying fully typed programs. Types are already available. We just check if the type rules are obeyed.
- *Type inference* is the process of filling in missing type information.
- Type-checking and type-interference are different terms, but people often use them interchangeably.

Type checking formalism

- Inference rules have the form
    - If Hypothesis is true, then Conclusion is true
- Type checking computes via reasoning
    - If E1 and E2 have certain types, then E3 has a certain type
- Rules of inference are a compact notation for "If-Then" statements
- The notation is easy to read with practice
- We will start with a simplified system and gradually add features
- Building blocks
    - Symbol ^ is "and"
    - Symbol => is "if-then"
    - x:T is "x has type T"
- If `e1` has type `Int` and `e2` has type `Int`, then `e1+e2` has type Int
- `(e1:Int ^ e2:Int) => e1+e2:Int`
- `(e1:Int ^ e2:Int) => e1+e2:Int` is a special case of `Hypothesis1 ^ Hypothesis2 ... ^ HypothesisN => Conclusion` This is an inference rule
- By tradition inference rules are written

```
|- Hypothesis ... |- Hypothesis
-------------------------------
     |- Conclusion
```

- e.g., C types have hypotheses and conclusions of the form

```
|- e:T
```

- `|-` means "it is provable that ..."

Some rules

```
i is an integer literal (constant)
----------------------------------  [Int]
     |- i:int

|-e1:int    |-e2:int
----------------------------------  [Add]
     |-e1+e2:int
```

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions

```
1 is an int literal
-------------------
  |- 1:int

2 is an int literal
-------------------
  |- 2:int

|-1:int   |-2:int
-----------------
   |-1+2:int
```

- A type system is *sound* if
    - Whenever `|-e:T`
    - Then `e` evaluates to a value of type `T`
- We only want sound rules
    - But some sound rules are better than others!

    ```
    1 is an integer literal
    -----------------------
       |-1:void
    ```

    This is correct (sound) but not the most precise typing rule. e.g., it will not allow 1 to be added because voids cannot be added. assumption: `int` is a subtype of `void`

Type Checking

- Type checking proves facts `e:T`
    - Proof is on the structure of the AST
    - Proof has the shape of the AST
    - One type rule is used for each AST node
- In the type rule used for a node `e`:
    - Hypotheses are the proofs of types of `e`'s subexpressions
    - Conclusion is a type of `e`
- Types are computed in a bottom-up pass over the AST

Type Environments

```
------------------  [false]
  |- false : bool

 s is a character literal
------------------------- [char]
  |- s : char

   |-s:T
----------------------- [address-of]
  |- &s : T *

 s is a string literal
----------------------- [string]
  |- s : char *

    |-b:bool
----------------------- [Not]
  |- !b : bool

    |-x:T
----------------------- [Expr-Stmt]
  |- x=y; : T

 |-x:T  |-y:T
----------------------- [Assignment-Stmt]
  |- x=y : T

 |-e:bool  |-x:T
------------------------- [While-Stmt]
  |- while (e) x : void
```

This is a design decision. One could have said that the type of a while loop is the type of the last statement that executes. But then what if the statement didn't execute even once? Hence a type 'void' means that the statement has a type which is not usable. Similarly for if-then-else: the types in the two branches can be different, so the final type is void.

But what is the type of a variable reference?

```
   x is a variable
  ----------------  [Var]
      |- x:?
```

The local, structural rule does not carry enough information to give `x` a type.

Solution: put more information in the rules!

A *type environment* gives types for *free* variables

- A type environment is a function from ObjectIdentifiers to Types
- A variable is free in an expression if it is not defined within the expression
    - `x` is free in expression `x`
    - `x` is free in expression `x+y`
    - `x` is free in expression `{int y = ...; x+y;}`
    - `y` is a *bound variable* in expression `{int y = ...; x+y;}`

The sentence `O |- e:T` is read

- Under the assumption that free variables have the types given by O, it is provable (|-) that expression `e` has the type `T`

```
i is an integer literal (constant)
----------------------------------  [Int]
    O |- i:int

 O |- e1:int    O |- e2:int
----------------------------------  [Add]
    O |- e1+e2:int
```

Notice that the same assumptions are required in both the hypotheses and the conclusion.

And we can write new rules

```
O(x) = T
---------  [Var]
O |- x:T
```

```
      O[T0/x] |- e1:T1
----------------------------    [Let-No-Init]
 O |- { T0 x; e1 } : T1
```

O[T/x] represents one function, where

```
O[T/x](x) = T
O[T/x](y) = O(y)
```

When we enter the scope, we add a new assumption in our type environment. When we leave the scope, we remove the assumption.

This terminology reminds us of the symbol table. So the type-environment is stored in the symbol table.

- The type environment gives types to the free identifiers in the current scope
- The type environment is passed down the AST from the root towards the leaves
    - To check the type of a new scope, we will see which rules can be applied
    - The Let-Init/Let-No-Init rule will indicate that we should try type-checking the body of the new scope with an updated environment (top-down): O[T/x]
- Types are computed up the AST from the leaves towards the root.

```
      O[T0/x] |- e1:T1
         O |- e2:T0
----------------------------    [Let-Init]
 O |- { T0 x = e2; e1 } : T1
```

Notice that this rule says that the initializer `e2` should have the same type `T0` as `x`. This is quite weak. In general, we should allow such assignment as long as `e2` is of a subtype of `T0`.

Define a relation <= on C++ classes

- X <= X
- X <= Y if X inherits from Y
- X <= Z if X <=Y and Y <= Z

A better version of [Let-Init] using subtyping:

```
      O[T0/x] |- e1:T1
         O |- e2:T0
       T0 <= T
----------------------------    [Let-Init]
 O |- { T x = e2; e1 } : T1
```

A better version of [Assignment-Stmt] using subtyping

```
 O|-x:T0  O|-e1:T1 T1 <=T0
------------------------------ [Assignment-Stmt]
  |- x=e1 : T0
```

(Notice that the type of the assignment statement is T1, so it can participate in more operations than T0)

X?y:z with subtyping. define least-upper-bound. use least-upper-bound to type this ternary operator. Compare with if-then-else

Typing Methods

C++ example

```
        O |- e1:T1
        ...
        O |- en:T1
------------------------------ [Dispatch]
   O |- f(e1,e2,...,en):?
```

Also maintain a mapping for method signatures in O

```
O(f) = (T1,...,Tn,T(n+1))
```

means that there is a method f such that

```
f(x1:T1,x2:T2,x3:T3,...,xn:Tn):T(n+1)
```

Hence, our dispatch rule becomes:

```
        O |- e1:T1
        ...
        O |- en:Tn
        O |- f:(T1',..,Tn',T(n+1)')
        T1<=T1',...Tn<=Tn'
---------------------------------------- [Dispatch]
   O |- f(e1,e2,...,en):T(n+1)
```

Object-oriented languages also implement encapsulation (e.g., some functions are only visible in certain classes, etc.). To handle them, one can extend O to be a function from a tuple of class-name and the identifier-name. Further, to express inheritance, we will use the subtype-relation for the "this" object in the method dispatch.

```
        O |- e0:T0
        O |- e1:T1
        ...
        O |- en:Tn
        O |- C::f : (T1',..,Tn',T(n+1)')
        T0 < T
        T1<=T1',...Tn<=Tn'
---------------------------------------- [Static-Dispatch]
   O |- e0@T.f(e1,e2,...,en):T(n+1)
```

For object-oriented languages, you also need to know the "current class" C in which the expression is sitting.

The form of a *sentence* in the logic is:

```
O,C |- e:T
```

E.g.,

```
 O,C |- e1:int    O,C |- e2:int
----------------------------------  [Add]
    O,C |- e1+e2:int
```

General themes

- Type rules are defined on the structure of expressions (through structural induction)
- Types of variables are modeled by an environment

Warning: Type rules are very compact! A lot of information in compact rules and it takes time to interpret them.

Implementing Type Checking

- C type checking can be implemented in a single traversal over the AST. This is not true for some other languages like OCaml for example, where multiple passes may be required.
- Type environment is passed down the tree
    - From parent to child
    - Other languages may require multiple passes to construct the type environment at each node
- Types are passed up the tree
    - From child to parent

Let's consider the following rule:

```
 O,C |- e1:int    O,C |- e2:int
----------------------------------  [Add]
    O,C |- e1+e2:int
```

This says that to type-check `e1+e2`, then we have to type-check `e1` and `e2` separately in the same O,C environment.

```
TypeCheck(Environment, e1+e2) = {
  T1 = TypeCheck(Environment, e1);
  Check T1 == int;
  T2 = TypeCheck(Environment, e2);
  Check T2 == int;
  return int;
}
```

Let's consider a more complex example:

```
         O |- e0:T0
      O[T0/x] |- e1:T1
       T0 <= T
----------------------------    [Let-Init]
 O |- { T x = e0; e1 } : T1
```

Let's look at the corresponding type-checking implementation:

```
TypeCheck(Environment, { T x = e0; e1 }) = {
  T0 = TypeCheck(Environment, e0);
  T1 = TypeCheck(Environment.add(x:T), e1);
  Check subtype(T0,T1);
  return T1;
}
```

Static vs. Dynamic Typing

- Static type systems detect common errors at compile time
- But some correct programs are disallowed
    - Some argue for dynamic type checking instead. e.g., Python, Perl, ..
    - Others want more expressive static type checking. e.g., fancier and fancier type systems. But can become quite complex making it harder for programming with such highly-expressive type-systems.

  The dynamic type of an object is the class C that is used in the "`new C`" expression that created it.
    - A run-time notion
    - Even languages that are not statically typed have the notion of dynamic type
- The static type of an expression captures all dynamic types the expression could have
    - A compile-time notion

Formalizing the relationship:

Soundness theorem: for all expressions E, dynamic_type(E) = static_type(E)

In all executions, E evaluates to values of the type inferred by the compiler. You have to actually run the program to get the dynamic_type. In the early days of programming languages, this was the type of property that was proved for their type systems.

Consider C++ program:

```
class A { ... }
class B : public A { ... }
void foo() {
  A foo;
  B bar;
  ....
  foo =  bar
}
```

Static type of foo is "A", but the dynamic type of foo is "A" and "B" at different program points.

Soundness theorem for object-oriented languages that allow subtyping.

```
\forall{E}{dynamic_type(E) <= static_type(E)}
```

All operations that can be used on an object of type C can also be used on an object of type C' <= C.

- Such as fetching the value of an attribute
- Or invoking a method on the object

Subclasses *only add* attributes or methods (they never remove attributes/methods).

The dynamic type is obtained through the execution semantics of the program, e.g., if we have an expression `x++`, and the current state of the program determines `x` to be of type `int`, there is a dynamic typing rule that says that the type of the new expression is also `int`. Consider the example `{ T x = e0; x; }` where `e0:T0 <= T`; here the dynamic type of the expression is `T0` (not `T`!). Similarly, the dynamic type of `if x then y:T1 else Y:T2` will be either `T1` or `T2` (not `lub(T1,T2)` or `void`).

In other words, soundness states that the static type system will not accept any incorrect program (that will not pass the dynamic type check of equal power). A dynamic type check for array-bounds is not of equal power (it has more checks enabled); a sound static type check for array-bounds would reject all incorrect programs (but it will also reject a few more that will actually pass the dynamic type check).

Soundness of static type system: all dynamically-type-incorrect programs will be rejected. Ensured by ensuring that static_type is a supertype of dynamic_type. A trivial static-type system that rejects all programs is sound (but not useful).

Completeness: all dynamically-type-correct programs will be accepted. Not possible to ensure in general.

Methods can be redefined but with same type! e.g., C++, Java override.

Example where static type system can be too restrictive

While a static type system may be sound, it may be too restrictive and may disallow certain programs that may be dynamically well-typed.

```
class Count {
  int i = 0; //default value = 0
        Count inc() { i <-- i + 1; return *this; }
};
```

Now consider a subclass `Stock` of `Count`:

```
class Stock : public Count {
  string name; //name of the item
};
```

And the following use of `Stock`:

```
Stock a;
Stock b = a.inc();
... b.name ...
```

This code does not type-check because the return value of `inc` is `Count` which is not a subtype of `Stock`.

This seems like a major limitation as now the derived classes (subtypes) will be unable to use the `inc` method.

`a.inc()` has *dynamic type* `Stock`.

So it is legitimate to write: `Stock b <-- a.inc();`

But this is not well-typed

- `a.inc()` has *static type* `Count`.

- We can extend the type system. Different languages extend the type system in different directions

- Option 1: Use `dynamic_cast`: returns nullptr at runtime if not-successful; else returns a pointer to the object of the new type. Allows bypassing the static type system. May entail runtime cost.
- Option 2: C++ Template : pass type as argument

    -
      ```
      template<typename T>
      class Count<T> {
        int i = 0;
              T inc() { i <-- i + 1; return *static_cast<T *>(this); } //static_cast gets checked at compile-time!
      }

      class Stock : public Count<Stock>
      {
        ...
      }
      ```
    - Another option: provide `dynamic_cast` operator. Will fail at runtime if the cast cannot be executed successfully. The compiler just treats this as an operator that either returns `nullptr` (cast not successful) or (a pointer to) an element of the new type. In both cases, the compiler can perform the rest of the static type-check using the new type.
- Accept more correct programs
- Increase the expressive power of the type system

Error Recovery

- Detecting where errors occur is easier than in parsing
- Introduce a new type `No_type` for use with ill-typed expressions
- Define `No_type <= C` for all types `C`. Avoids cascading type errors due to one type-error.
- Thus, every operation is defined for `No_type`
    - With a `No_type` result
- The type hierarchy is not a tree anymore, it is a DAG with `No_type` at the bottom