Compilers Course
Lecture 9: Semantic Analysis

Introduction:

The purpose of semantic analysis is to verify "static" correctness, i.e. to detect invalid programs:

- Uses of identifiers must be consistent with their declarations
- The right number and types of parameters must be passed to procedures and operators
- Other language-specific rules for example, if a Java compiler cannot prove that a variable is initialized before being used, then the compiler must reject the program as being invalid

Some things are typically not checked in the compiler's static analysis:

- Division by zero
- Out of bounds indexes in arrays
- NULL pointers being dereferenced

It is usually impossible to prove the absence of these errors at compile-time, because they depend on runtime data. So most compilers either don't check them at all (C), or generate code to check them at runtime (Java, ML).

Tools for semantic analysis

Type Systems:

- Describes the types available in the language (int, boolean, real, string, arrays, functions, etc.)
- Describes the rules for how types may be combined in various language constructs (adding two int:s results in an int, comparing two numbers (int or real) results in a boolean, etc.)

Environments / Symbol tables:

- Records semantic information (usually types) about identifiers
- Handles scope (visibility) rules

We usually use the term "environment" when dealing with semantics in an abstract sense, and "symbol table" when dealing with a concrete data structure in a compiler.

Scope (visibility) rules

A language's scope rules determines how uses of identifiers are connected to declarations (bindings) of those identifiers.

There are two main kinds of scope rules: static scoping (common) and dynamic scoping (obsolete).

Static (lexical) scoping

- A binding is visible from its declaration and forward in the same scope (leads to the familiar

declare-before-use rule)
- In some languages a binding may be visible from the start of the enclosing scope, e.g. methods in a class may be defined in any order
- Scopes can be nested: a function body is a nested scope, and blocks ("{" ... "}" in C) introduce new nested scopes
- Bindings in outer scopes are visible in inner scopes, but not vice-versa
- Bindings in nested scopes can override (shadow) bindings in outer scopes

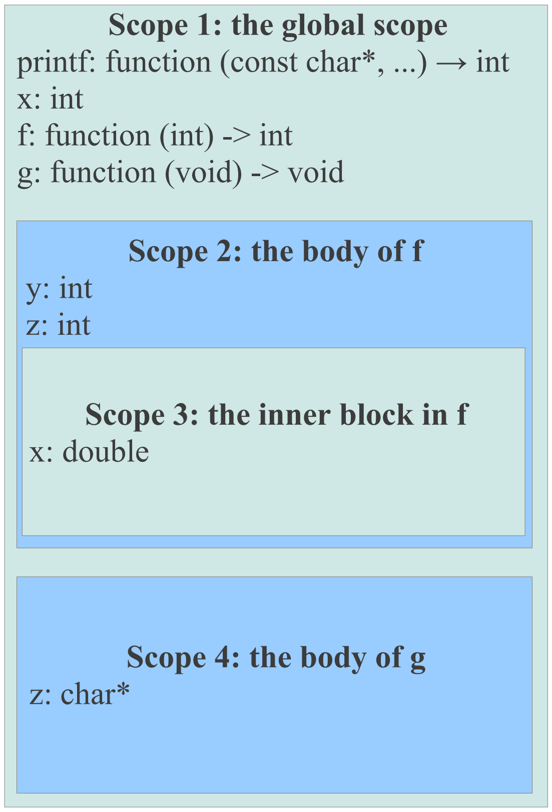
Most of the current programming languages use static scoping.

Example in C:

```
int printf(const char*, ...);
int x = 5;

void f(int y)
{
    int z = x;
    {
        double x = 3.14;
        printf("%f", z+x);
    }
}

void g(void)
{
    char *z = "hello";
    printf("%s", z);
}
```


Scope 1: the global scope
printf: function (const char*, ...) → int
x: int
f: function (int) -> int
g: function (void) -> void
Scope 2: the body of f
y: int
z: int
Scope 3: the inner block in f
x: double
Scope 4: the body of g
z: char*

Dynamic scoping

Consider this example in C:

```
int x = 5;
int f() { return x; }
int main(void) { int x = 0; return f(); }
```

With static (lexical) scoping, the "x" in the body of "f" always refers to the global "x" (5).

With dynamic scoping, a variable always refers to the latest binding of that variable *in the execution order*. In the example above, main() binds x to zero, then calls f(), which returns the latest x, i.e. 0.

Dynamic scoping has many problems:

- Impossible to predict what values non-local variables will have
- Static typing becomes impossible

<u>**Using Attribute Grammars to describe type checking**</u>

To perform type checking in a procedural language we need:

- Types: int, boolean, real, string, etc
- Environments: mappings from identifiers to types

We will assume that operations for types and environments exist.

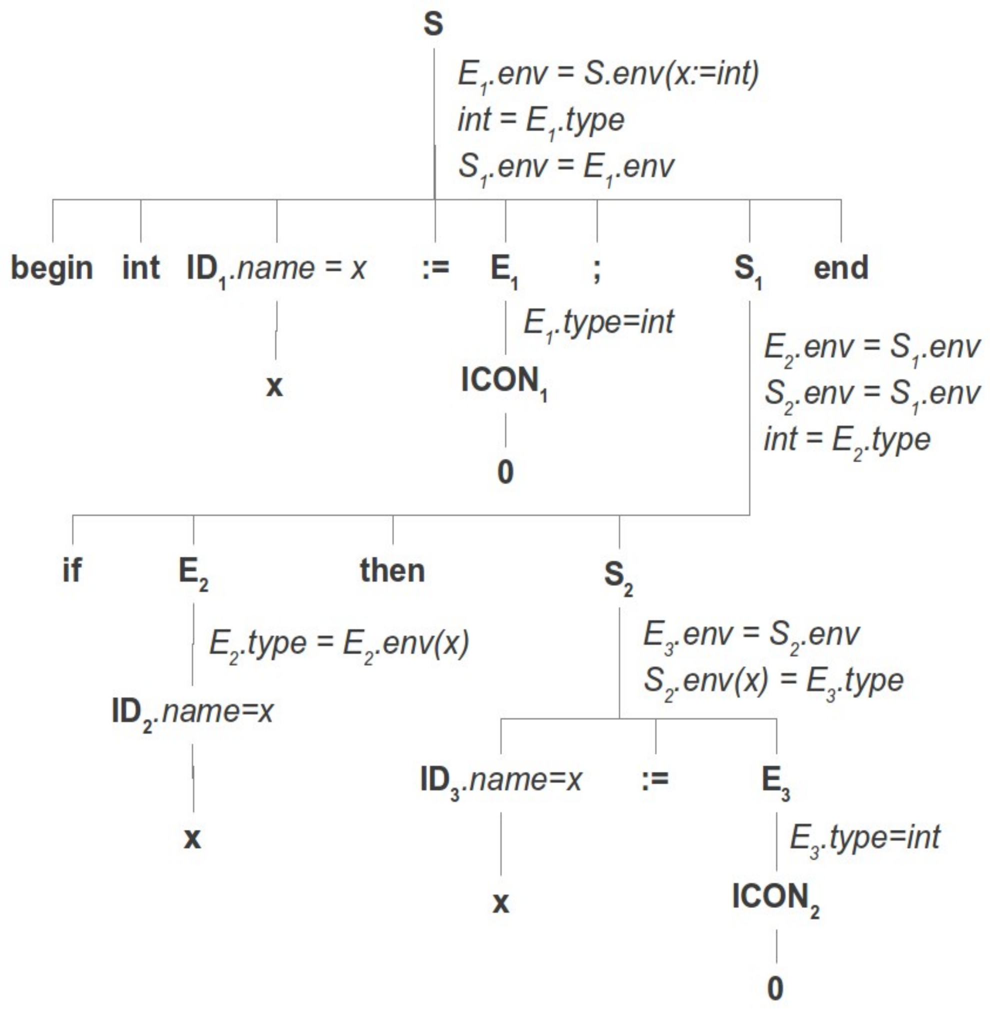
Statements and expressions will need an inherited "env" attribute for the types of free (not locally bound) identifiers.

Expressions will need a synthesized "type" attribute describing their types.

ID.name: string
E.env: environment
E.type: types
S.env: environment

```
E → ICON
        E.type = int
E → ID
        E.type = E.env(ID.name)
E → E1 + E2
        E1.env = E.env
        E2.env = E.env
        E.type =
          case (E1.type, E2.type)
            of (int, int) → int
             | _ → error
S → if E then S1
        E.env = S.env
        S1.env = S.env
        int = E.type     // a condition
S → begin int ID = E; S1 end
        E.env = S.env(ID.name := int)
        int = E.type     // a condition
        S1.env = E.env
S → ID := E
        E.env = S.env
        S.env(ID.name) = E.type       // a condition
```

begin int x := 0; if x then x := 0 end



Type-checking AGs like this one are usually L-attributed, so they require only a single left-to-right scan of the syntax tree.

We will talk more about type checking later.

Symbol Table:

A binding is an association of a *key* with some *value.* An environment is a dynamic set of bindings. Typical operations on environments include:

- Create an empty environment
- Insert a binding into an environment
- Check if a key is bound in the environment

- Look up a key's value in the environment
- Add all bindings in one environment to another

A compiler written in an imperative language usually implements its type checking environment as a single hash table with chaining:

- Each hash table element is a structure containing
    - Key (e.g. a name)
    - Hash code (optional, used by resizable hash tables)
    - Pointer to the next element on the collision chain (same hash code)
    - Some data (e.g. a type descriptor)
- The hash table is an array of collision chains (pointers to hash table elements)
- Hash(name) maps name to a non-negative integer
- Lookup(name):
```
  index = hash(name) mod hash_array_size
  list = hash_table[index]
  loop over list, compare names, return first element that matches
```
- Insert(name):
```
  index = hash(name) mod hash_array_size
  list = hash_table[index]
  allocate a new hash table element
  new_element.name = name; new_element.next = list
  hash_table[index] = new_element
```

A hash table is a simple well-known structure that offers near constant-time insertions and lookups.

Nested Scopes: The Problem

- As declarations in a scope are processed, they are added to the hash table.
- Nested scopes may redefine identifiers bound in outer scopes, and thus already present in the hash table.
- When the compiler is finished with a nested scope, the hash table *must* be restored to the state it had before the compiler started processing that nested scope.

Nested Scopes: Solutions

1. Stack of hash tables, one per scope.
2. Copy hash table before nested scope, restore afterwards.
3. Repair hash table when leaving nested scope, by enumerating the identifiers declared in it.
    - Mark bindings with scope nesting level.
    - Record a scope's bindings in a separate per-scope set.
    - Traverse AST again.

Alt 1: stack of hash tables
Let each scope have its own hash table, then organize the hash tables in a stack: search top table first, then the second, and so on. At the end of a scope, remove top table.

Problems:

- With deep nesting lookups degenerate to linear searches
- Must handle both small scopes and large scopes well, which requires hash tables with automatic resizing (more complex)
- Needs more space for N hash tables than one large hash table

Alt 2: copy hash table before entering a scope
At the start of a nested scope, make a copy of the outer scope's hash table. Do insertions and lookups using the copy. At the end of the scope, discard the copy and restore the previous hash table.

Problems:

- Must copy the index array
- Must copy the chains too if the hash table can be resized (resizing rearranges the chains)
- Expensive in time and memory space

Alt 3a: make the hash table remember each scope's set of bindings using scope numbers

- Add scope level number to each hash table entry.
- Every insertion in the current scope also stores current scope level in the entry.
- Add begin_scope operation: increments the scope level.
- Add end_scope operation: loops over all collision chains and removes elements having the current scope level, then decrements the scope level.

  + Simple solution
  - Insert() *must* push new bindings at the heads of the collision chains, so that when some identifier has multiple bindings the most recent one is found first; this is common practice anyway but now it's crucial
  - End_scope is potentially very expensive

Alt 3b: make the hash table remember each scope's set of bindings using a stack of lists

- Every insertion in the current scope is recorded in a list, using a stack of lists for the nested scopes.
- Add begin_scope operation: pushes an empty list on the scope stack.
- Add end_scope operation: pops the top-most list off the scope stack, every element in the list is then removed from the hash table.

  + Simple solution
  - Insert() *must* push new bindings at the heads of the collision chains
  - Duplicates the contents of every scope in a third structure:
  1: AST, 2: hash table, 3: the new per-scope list/set

Alt 3c: let clients keep track of their scopes
The user of the symbol table (e.g. the compiler's type checker) must process each scope twice:

- First traverse the scope's declarations (the syntax tree) and enter them in the hash table.
- Type-check the statements/expressions inside the scope.
- Before leaving the scope, traverse the scope's declarations again and remove them from the hash table.

  + Simplifies the symbol table.
  + Avoids the per-scope list/set.
  - Requires new "remove element" operation for the symbol table.
  - Requires the type checker to regenerate the current scope's symbols before leaving the scope, which requires some work. For example, processing C declarations requires a recursive procedure.
  - Like in Alt 3a and 3b insert() must add new bindings at the heads of the collision chains.

Alt 3b with a scope stack is probably the best solution in terms of simplicity and efficiency.
Alt 3c can use a little less memory, but is more complex.
Alt 1 may be OK in a toy compiler.

Checking redefinition errors in current scope

A nested scope can shadow bindings in outer scopes, but a particular scope is usually not allowed to bind the same variable more than once.

Alternative 1:

- Build miniature symbol table with just the new scope's bindings
- Check for duplicates in this mini-table.
- Finally add mini-table to the actual symbol table.

  - Requires additional symbol table like structure with potentially different optimization requirements
  - Temporary memory costs for the mini-table

Alternative 2:

- Add scope level number to bindings, and a global scope level variable.
- Increment the scope level at start of a scope, and decrement it at the end of the scope.
- When binding an identifier, first try to look it up. If it is bound, verify that its scope level is less than the current scope level. Then proceed and insert the new binding.

  + Simple
  - Permanent memory cost since every binding must also have a scope number

Symbol Table Implementation: Binary Trees

With binary trees insertions and lookups become O(log N) rather than O(1). So why use binary trees?

- Compilers written in functional languages *cannot* use side-effects to update hash tables, so every update would require the index array to be copied, which has O(N) cost.
- Updating a binary tree without modifying the original tree requires the path to the updated node to be copied, which has O(log N) cost.
- Handling nested scopes is extremely easy:
    - At the beginning of a scope, remember the current value of the tree: O(1) cost to bind it to a variable.
    - At the end of a scope, forget about the nested scope's tree and use the remembered one instead: O(1) cost. (Assumes a garbage collector will reclaim the memory used for the discarded tree.)

Compilers written in functional languages tend to prefer binary trees over hash tables.

Name Spaces

A programming language has different kinds of identifiers:
- Variables
- Functions/procedures
- Labels for goto
- Type names
- Fields in records

These exist in different name spaces:
- Record fields are private to that record type
- Type names are often separate from variable and function names

In C, variables, functions, and type names (builtin and *typedef*) share the same name space. Names for struct/union/enum types are in a separate name space.

Implementation:
- Different symbol tables for different name spaces.
- Shared symbol table, using <name, name space tag> pairs as keys.
- The fields of a record are usually recorded in a list (mini-table) embedded in the representation of the record type.

Type Checking

Types

A runtime type is a set of values (bit patterns) and operations defined on those values. Ex: 01 + 01 → 10

A compile-time type is an abstract description of a runtime type. Ex: int is 32 bits, int + int → int

A source code type is a language construct that maps to a compile-time type. Ex: "int" → int

The compile-time type checker:

- Analyses the program's declarations and associates compile-time types with all variables and functions
- Analyses the program's expressions and infers compile-time types for all expressions
- Verifies that sub-expressions are used correctly in surrounding expressions (E_1 *op* E_2) and statements (var := E, if/while, return E)

The type checker is like an interpreter for the language, that computes with compile-time types rather than runtime values.

Type system

- Set of base types (int, boolean, real, etc)
- Rules for forming composite types (arrays, pointers, records, functions, etc)
- Rules for type inclusion (subtyping, automatic conversion) *common* for all operations:
  char < short < int < long *[in C]*
  int < double *[in C]*
  nil < any pointer type *[in Pascal]*
  0 < any pointer type *[in C]*
  array of T < pointer to T *[in C]*
  This can be formalized as a lattice structure.
- Rules for combining types in specific language operations

Representing types

Types have a tree structure very similar to the structure of e.g. expressions: the leaves are primitive types, and internal nodes are composite types.

Types can be represented just like abstract syntax trees, e.g. in SML:

```
datatype ty = INT | BOOL | REAL
                    | ARRAY of ty * int
                    | POINTER of ty
                    | FUN of ty * ty list
```

```
                        | STRUCT of (ident * ty) list
```

or in C:

```
enum base_type { INT, BOOL, REAL };
enum ty_tag { BASE, ARRAY, POINTER, ... };
struct ty {
    enum ty_tag tag;
    union {
        enum base_type base;
        struct {
            struct ty *element_type;
            int size;
        } array;
        struct ty *pointer;
     struct {
         struct ty *res;
         struct ty_list *args;
     } fun;
        ...
    } u;
};
struct ty_list {
    struct ty *first;
    struct ty_list *rest;
};
```

Type Checking Expressions

A recursive function that performs a preorder traversal of the AST:

- Determine the top-level structure of the expression
- Recursively type-check ("evaluate") sub-expressions
- Finally check that sub-expressions and operators are used correctly:
    - "VAR": look it up in the symbol table
    - "CONST": known type depending on what kind of constant it is
    - "E1 ^ E2": both must be integers, result is integer
    - "E1 + E2": both must be integers or floats, the result is the larger of the types
    - "E1 > E2": both must be integers or floats, result is integer
    - "A[E]": A must be an array or pointer, E must be an integer, result is A's element type
    - "f(E_1,..., E_n)": f must be a function with n parameters, each Ei (actual parameter) must be assignment-compatible with the corresponding formal parameter, result is f's return type

    - Return the type of the expression

The rules for type-checking full C are much more detailed than this, they involve implicit widening of small numerical types, taking the least upper bound of two numerical types, and handling C-specific features like pointer arithmetic and the NULL pointer constant.

Type Checking Statements

Similar to expressions, except statements do not have types so only return OK or Error.

Some specific conditions to check:

- "if (E)": E must have boolean or integer type
- "while (E)": *ditto*
- "VAR := E": VAR must be declared, and E must be "assignment-compatible" with VAR:
  ty1 := ty1 is ok by definition
  ty1 := ty2 is ok if ty2 < ty1 ("widening"), or if ty2 may be coerced ("narrowed") to ty1 by an assignment. In C, long integers may be narrowed to smaller integers, and floats may be truncated to integers.
- "return E": treated like an assignment to a variable with the same type as the current function's return type

"Type Checking" Declarations

The main purpose here is to analyze the declarations and add information to the symbol table.

- Process top-level declarations in program order
- For variables:
    - Check that the variable is not already defined in the current scope
    - Evaluate the type expression
    - Then bind the variable to that type in the symbol table
- For functions:
    - Check that the function is not already defined in the current scope
    - Evaluate the result and argument types, and bind the function to that function type in the symbol table
    - Type-check the function's body (compound statement)
- For compound statements:
    - Create a new scope
    - Process declarations (formal parameters + local variables):
        - Check that no identifier is multiply defined in this scope
        - Bind the identifiers to their types in the symbol table
    - Type-check the statements
    - Leave the scope, restoring the previous symbol table

Issues in other languages

This view of type checking is appropriate for simple procedural languages like C and Pascal. Other languages may require extensions:

1. Overloaded methods in C++ and Java. Ex: *int f(int x); double f(double x);* There may be multiple declarations for a given method. At a call the compiler must search for the *most appropriate* version given the actual types of the arguments in the call.
2. Type inference in SML. Ex: *fun f(x) = x+1* The type checker must operate on types with partially unknown structure. Then it must be able to incrementally refine that structure as requirements are detected. Typically implemented using "type variables" and "unification" of type structures.