The Compiler So Far

- **Scanner - Lexical analysis**
  - Detects inputs with illegal tokens
  - e.g.: main 5 (;)
- **Parser - Syntactic analysis**
  - Detects inputs with ill-formed parse trees
  - e.g.: missing semicolons
- **Semantic analysis**
  - Last "front end" analysis phase
  - Catches all remaining errors

---

What's wrong with this code?

(Note: it parses perfectly)

```
foo(int a, char * s){ ... }
int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s\n",p,q);
  goto label23;
}
```

---

Goals of a Semantic Analyzer

- Compiler must do more than recognize whether a sentence belongs to the language...
- Find remaining errors that would make program invalid
  - undefined variables, types
  - type errors that can be caught statically
- Figure out useful information for later phases
  - types of all expressions
  - data layout

Terminology

- Static checks – done by the compiler
- Dynamic checks – done at run time

Kinds of Checks

- **Uniqueness checks**
  - Certain names must be unique
  - Many languages require variable declarations
- **Flow-of-control checks**
  - Match control-flow operators with structures
  - Example: break applies to innermost loop/switch
- **Type checks**
  - Check compatibility of operators and operands

Logical checks

- Program is syntactically and semantically correct, but does not do the "correct" thing
Examples of Reported Errors

- Undeclared identifier
- Multiply declared identifier
- Index out of bounds
- Wrong number or types of args to call
- Incompatible types for operation
- Break statement outside switch/loop
- Goto with no label

Program Checking

- Why do we care?
  - Obvious:
    - Report mistakes to programmer
    - Avoid bugs: \texttt{f[6] will cause a run-time failure}
    - Help programmer verify intent
  - How do these checks help compilers?
    - Allocate right amount of space for variables
    - Select right machine operations
    - Proper implementation of control structures

Can We Catch Everything?

- Try compiling this code:
  ```c
  void main()
  {
    int i=21, j=42;
    printf("Hello World\n");
    printf("Hello World, N=%d\n", i, j);
    printf("Hello World, N=%d\n");
    printf("Hello World, N=%d\n");
  }
  ```

Inlined TypeChecker and CodeGen

- You could type check and generate code as part of semantic actions:
  ```c
  expr : expr PLUS expr {
    if ($1.type == $3.type &&
        ($1.type == IntType ||
         $1.type == RealType))
      $$.type = $1.type
    else error("+ applied on wrong type!");
    GenerateAdd($1, $3, $$);
  }
  ```

Problems

- Difficult to read
- Difficult to maintain
- Compiler must analyze program in order parsed
  - Instead ... we split up tasks

Compiler ‘main program’

```c
void Compile() {
  AST tree = Parser(program);
  if (TypeCheck(tree))
    IR ir =
      GenIntermedCode(tree);
    EmitCode(ir);
}
```
Typical Semantic Errors

- **Multiple declarations**: a variable should be declared (in the same scope) at most once
- **Undeclared variable**: a variable should not be used before being declared
- **Type mismatch**: type of the LHS of an assignment should match the type of the RHS
- **Wrong arguments**: methods should be called with the right number and types of arguments

A Sample Semantic Analyzer

- Works in two phases — traverses the AST created by the parser
  1. For each scope in the program
     - process the declarations
       - add new entries to the symbol table and
       - report any variables that are multiply declared
     - process the statements
       - find uses of undeclared variables, and
       - update the "ID" nodes of the AST to point to the appropriate symbol-table entry.
  2. Process all of the statements in the program again
     - use the symbol-table information to determine the type of each expression, and to find type errors.

Scoping

- In most languages, the same name can be declared multiple times
  - if its declarations occur in different scopes, and/or
  - involve different kinds of names
- Java: can use same name for
  - a class
  - a method of the class
  - a local variable of the method

```java
class Test {
    int Test;
    void Test( ) { double Test; }
}
```

Scoping: Overloading

- Java and C++ (but not in Pascal or C):
  - can use the same name for more than one method
  - as long as the number and/or types of parameters are unique

```java
int add(int a, int b);
float add(float a, float b);
```

Scoping: General Rules

- The scope rules of a language:
  - Determine which declaration of a named object corresponds to each use of the object
  - Scoping rules map uses of objects to their declarations

- C++ and Java use static scoping:
  - Mapping from uses to declarations at compile time
  - C++ uses the "most closely nested" rule
  - a use of variable x matches the declaration in the most closely enclosing scope
  - such that the declaration precedes the use

Scope levels

- Each function has two or more scopes:
  - One for the function body
  - Sometimes parameters are separate scope!
    - (Not true in C)
      ```c
      void f( int k ) { // k is a parameter
        int k = 0; // also a local variable
        while (k) {
          int k = 1; // another local var, in a loop
        }
      }
      ```
  - Additional scopes in the function
    - each for loop and
    - each nested block (delimited by curly braces)
**Checkpoint #1**

- Match each use to its declaration, or say why it is a use of an undeclared variable.

```
int k=10, x=20;
void foo(int k) {
    int a = x; int x = k; int b = x;
    while (...) {
        int x;
        if (x == k) {
            int k, y;
            k = y = x;
        }
        if (x == k) { int x = y; }
    }
}
```

**Dynamic Scoping**

- Not all languages use static scoping
- Lisp, APL, and Snobol use dynamic scoping

- Dynamic scoping:
  - A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the most-recently-called still active function

**Example**

- For example, consider the following code:

```
int i = 1;
void func() {
    cout << i << endl;
}
int main () {
    int i = 2;
    func();
    return 0;
}
```

If C++ used dynamic scoping, this would print out 2, not 1

**Checkpoint #2**

- Assuming that dynamic scoping is used, what is output by the following program?

```
void main() { int x = 0; f1(); g(); f2(); }
void f1() { int x = 10; g(); }
void f2() { int x = 20; f1(); g(); }
void g() { print(x); }
```

**Symbol Tables**

- Purpose:
  - keep track of names declared in the program
- Symbol table entry:
  - associates a name with a set of attributes, e.g.:
    - kind of name (variable, class, field, method, ...)
    - type (int, float, ...)
    - nesting level
    - mem location (where it will be found at runtime)
- Functions:
  - Type Lookup(String id)
  - Void Add(String id, Type binding)
- Bindings: name type pairs (a → string, b → int)
Environments

• Represents a set of mappings in the symbol table

function f(a:int, b:int, c:int) =
  print_int(a+c);
  let var j := a+b
  var a := "hello"
  in print(a); print_int(j)
end;
print_int(b)
)

\[ \sigma_0 \]

\[ \sigma_1 = \sigma_0 + a \rightarrow \text{int} \]

\[ \sigma_2 = \sigma_1 + j \rightarrow \text{int} \]

Semantic Analysis

How Symbol Tables Work (1)

int x;
char y;
void p(void)
{ double x;
  { int y[10];
    ... }
  ...
}
void q(void)
{ int y;
  ... }
main()
{ char x;
  ... }

Semantic Analysis

How Symbol Tables Work (2)

int x;
char y;
void p(void)
{ double x;
  { int y[10];
    ... }
  ...
}
void q(void)
{ int y;
  ... }
main()
{ char x;
  ... }

Semantic Analysis

How Symbol Tables Work (3)

int x;
char y;
void p(void)
{ double x;
  { int y[10];
    ... }
  ...
}
void q(void)
{ int y;
  ... }
main()
{ char x;
  ... }

Semantic Analysis

How Symbol Tables Work (4)

int x;
char y;
void p(void)
{ double x;
  { int y[10];
    ... }
  ...
}
void q(void)
{ int y;
  ... }
main()
{ char x;
  ... }

Semantic Analysis

How Symbol Tables Work (5)

int x;
char y;
void p(void)
{ double x;
  { int y[10];
    ... }
  ...
}
void q(void)
{ int y;
  ... }
main()
{ char x;
  ... }

Semantic Analysis
How Symbol Tables Work

int x;
char y;
void p(void)
{
    double x;
    ...
}

void q(void)
{
    int y;
    ...
}

main()
{
    char x;
    ...
}

A Symbol Table Implementation

• Two structures: Hash table, Scope Stack
  • Symbol = foo
  • Hash(foo) = i

Symbol table

Semantic Analysis

Enter/Exit Scope

• We also need a stack to keep track of the “nesting level” as we traverse the tree...

Variables vs. Types

• Often, compilers maintain separate symbol tables for Types vs. Variables/Functions
  • Lecture Checkpoint:
    • ✓ Scopes
    • → Types

Types

• What is a type?
  – The notion varies from language to language

• Consensus
  – A set of values
  – A set of operations allowed on those values

• Certain operations are legal for 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!

Type Systems

• 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 the correct types
    – Enforces intended interpretation of values
  • Type systems provide a concise formalization of the semantic checking rules
Why Do We Need Type Systems?

- Consider the assembly language fragment
  \[ \text{addi} \; r1, \; r2, \; r3 \]

- What are the types of \( r1, \; r2, \; r3 \)?

Type Checking Overview

- Four kinds of languages:
  - **Statically typed**: All or almost all checking of types is done as part of compilation
  - **Dynamically typed**: Almost all checking of types is done as part of program execution (no compiler) as in Perl, Ruby
  - **Mixed Model**: Java
  - **Untyped**: No type checking (machine code)

Semantic Analysis

Type Checking and Type Inference

- **Type Checking** is the process of verifying fully typed programs
  - Given an operation and an operand of some type, determine whether the operation is allowed
- **Type Inference** is the process of filling in missing type information
  - Given the type of operands, determine
    - the meaning of the operation
    - the type of the operation
  - Or, without variable declarations, infer type from the way the variable is used
- The two are different, but are often used interchangeably

Issues in Typing

- Does the language have a type system?
  - Untyped languages (e.g. assembly) have no type system at all
- When is typing performed?
  - Static typing: At compile time
  - Dynamic typing: At runtime
- How strictly are the rules enforced?
  - Strongly typed: No exceptions
  - Weakly typed: With well-defined exceptions
- Type equivalence & subtyping
  - When are two types equivalent?
    - What does "equivalent" mean anyway?
    - When can one type replace another?

Components of a Type System

- Built-in types
- Rules for constructing new types
  - Where do we store type information?
- Rules for determining if two types are equivalent
- Rules for inferring the types of expressions

Component: Built-in Types

- Integer
  - usual operations: standard arithmetic
- Floating point
  - usual operations: standard arithmetic
- Character
  - character set generally ordered lexicographically
  - usual operations: (lexicographic) comparisons
- Boolean
  - usual operations: not, and, or, xor
Component: Type Constructors

- **Arrays**
  - array(I,T) denotes the type of an array with elements of type T, and index set I
  - multidimensional arrays are just arrays where T is also an array
  - operations: element access, array assignment, products
- **Strings**
  - bitstrings, character strings
  - operations: concatenation, lexicographic comparison
- **Records (structs)**
  - Groups of multiple objects of different types where the elements are given specific names.

Component: Type Constructors

- **Pointers**
  - addresses
  - operations: arithmetic, dereferencing, referencing
  - issue: equivalency
- **Function types**
  - A function such as "int add(real, int)" has type real*int→int

Component: Type Equivalence

- **Name equivalence**
  - Types are equiv only when they have the same name
- **Structural equivalence**
  - Types are equiv when they have the same structure
- **Example**
  - C uses structural equivalence for structs and name equivalence for arrays/pointers

Component: Type Equivalence

- **Type Coercion**
  - If x is float, is x=3 acceptable?
    - Disallow
    - Allow and implicitly convert 3 to float
    - "Allow" but require programmer to explicitly convert 3 to float
  - What should be allowed?
    - float to int ?
    - int to float ?
    - What if multiple coercions are possible?
      - Consider 3 + "4" ...

Formalizing Types: Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)
- The appropriate formalism for type checking is logical rules of inference

\[ \frac{e_1 : \text{int}}{e_2 : \text{int}} \frac{e_2 < e_3 : \text{boolean}}{e_3 : \text{int}} \]

Semantic Analysis Summary

- Compiler must do more than recognize whether a sentence belongs to the language
- **Checks of all kinds**
  - undefined variables, types
  - type errors that can be caught statically
- **Store useful information for later phases**
  - types of all expressions