Semantic Analysis

$$\frac{O \vdash e_1 : T \quad \vdash e_2 : T}{O \vdash e_1 \oplus e_2 : T}$$

Semantic Analysis

- (Static) semantic analysis: last phase of the frontend
  - catches remaining errors in the source code
  - typically errors depending on context

- Error detection stages
  - lexical analysis: invalid/malformed tokens
    - regular expressions
  - syntax analysis: invalid parse trees
    - context-free grammars
  - semantic analysis: illegal semantics
    - constructs with context

Semantic Analysis

- Typical checks performed during semantic analysis
  - declaration of identifiers
    - no reserved identifiers
    - definition before use
    - single definition
  - number of arguments in function calls
  - type checking
    - inheritance relationships
  - other constructs with context
    - program/procedure/function identifier
    - depend on the language

Scope

- Depending on the location ("scope") in a program, an identifier may refer to a different declaration

```
module test1;

var a,b: integer;

function foo(a: integer): integer;
begin
  return a + 1
end foo;

begin
  a := 1;
  b := foo(2);
  Output(a);
  Output(b)
end test1.
```

```
module test2;

var a: integer;

procedure foo(b: integer);
var a: integer;
begin
  a := b + 1
end foo;

begin
  a := 1;
  foo(a);
  Output(a)
end test2.
```

Scope

- The scope of an identifier is the portion of a program in which that identifier is accessible
  - different (non-overlapping) scopes
  - nested scopes

- Static vs. dynamic scope
  - static scope
    scope depends only on the source program, not the run-time behavior
    - Pascal, C, C++, Java, SnuPL/0

  - dynamic scope
    scope depends on the run-time behavior, i.e., execution of the program
    - older versions of Lisp, SNOBOL
    - recent languages do not use dynamic scope
    - exception: exception handlers

Static vs. Dynamic Scope

- Static vs. dynamic scope

```
module static;

var a: integer

function foo(a): integer;
begin
  return a + 1
end foo;

function bar(): integer;
begin
  return a + 1
end bar;

function foobar(): integer;
var a: integer;
begin
  return a + 1
end foobar;

begin
end static.
```

```
module dynamic;

function foobar(): integer;
begin
  return a + 1
end foobar;

function foo(): integer;
var a: integer;
begin
  a := 1;
  return foobar()
end foo;

function bar(): integer;
var a: integer;
begin
  a := 2;
  return foobar()
end bar;

begin
  Output(foo()); // prints 2
  Output(bar())  // prints 3
end dynamic.
```

Nested Scopes

- Scopes are perfectly nested
  - unique mapping of identifiers to variables
  - typically: most-closely-nested rule

```
var a: integer;

procedure foo(a);
var a: integer;
begin
   …
end foo;
```

Scope

- Examples

```
void foo()
{
  struct s { … };
  int a;

  for (int i=…) {
    if (i > N) {
      int a;
    }
  }
}
```

```
class A {
  public:
    int i;

    void foo(void) {
      int i;
    }

    virtual foobar() {
    }
  }
};

class B : public A {
  public:
    void bar(void) {
      i = …;
    }

    virtual foobar() {
    }
};
```

```
program A;

type
  TRec = record
    i: integer;
    f: float;
  end;

const
  i : integer = 5;

var
  a  : integer;

procedure foo(i: integer);
type PRec = ^TRec;
const f : float = 1.0;
var a: integer;

  procedure bar(i: integer);
  var a: integer;
  begin
  end;

var b: integer;
begin
end;

var b : integer
begin
end.
```

Symbol Tables

- Important data structure shared between the syntactic and semantic analysis (and likely later phases as well)
  - main purpose: map identifiers to storage locations
  - symbol information
    - identifier
    - type
    - kind
      - local, global, parameter
      - visibility (public, protected, …)
      - constant/variable
    - location
      - memory, stack (base + offset), register, …

Symbol Tables

- Stack-based implementation
  - add_symbol(x)
    push x on symbol stack along with necessary information
  - remove_symbol()
    pop the topmost symbol from the stack
  - find symbol(x)
    find symbol x, starting at the top of the stack

- Pros & Cons
  - simple
  - works for simple semantic analysis
  - no-go if the symbol table is to be used during later phases

Symbol Tables

- Hierarchical symbol tables managed by a symbol table manager
  - one symbol table per scope
    - perfectly nested scopes → perfectly nested symbol tables

  - enter_scope()
    begins a new scope
  - exit_scope()
    end a scope
  - add_symbol(x)
    add symbol x to the symbol table of the current scope
  - find_symbol(x, search_upwards)
    find symbol x in the current scope, if search_upwards==true, recurse upwards if not found
  - (alternatively) check_scope(x)
    check if symbol x is defined in the current scope

Symbol Tables

- Use before definition
  - typically not allowed
  - in certain programming languages OK
    - refer to classes/functions before use
  - two passes
    - pass 1: parse input, fill symbol tables with definitions
    - pass 2: parse bodies of scopes
  - Active Oberon compiler (ETHZ, successor to Oberon)
    - parallel parser
      - main thread only parses outermost scope
      - each subscope is parsed in parallel by a new thread
      - threads block on undefined symbols

Types

- The type of an identifier defines
  - a set of values (range)
  - a set of operations on those values

- Examples
  - integer
    - range: MIN_INT…MAX_INT
    - operations: +, -, *, /, MOD, REM, <, <=, ==, #, >=, >
  - boolean
    - range: TRUE, FALSE
    - operations: &&, ||, ==, #
  - string
    - range: 0 to n characters
    - operations: concatenation, length

Type System

- "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." – in *Types and Programming Languages* by Benjamin C. Pierce

- The type system of a language defines
  - basic types
    - range
    - operations
  - methods to build custom "base" types
    - enum
  - methods to build compound types
    - array, struct (record), union
  - type conversion rules
    - implicit
    - explicit

Type Checking

- Type checking ensures that operations are used only with correct types
  - machine language is untyped

```
int A[1024];
char s[32];
int i;

i = A + s;
```

```
leal A, %eax
leal s, %ecx
addl %eax, %ecx
movl %ecx, i
```

Type Checking

- Typed languages
  - Statically checked languages (aka “statically typed”)
    - (almost) all type checking done at compile type
    - Pascal, C, Java, SnuPL/1, …
  - Dynamically checked languages (aka “dynamically typed”)
    - (almost) all type checking done at run-time
    - Scheme, Lisp, Python, Perl, Javascript, …
- Untyped languages
  - machine code

Static vs. Dynamic Type Checking

- Static vs. Dynamic Type Checking
  - static type checking
    - catch many programming errors at compile time
    - avoid overhead of run-time checks
  - dynamic type checking
    - static type system is too restrictive
    - rapid prototyping difficult with a static type system
  - escape mechanisms on both sides
    - unsafe casts for statically-typed language
    - static typing for dynamically-typed languages for speed, debugging
  - debate still going on

Strong vs. Weak Typing

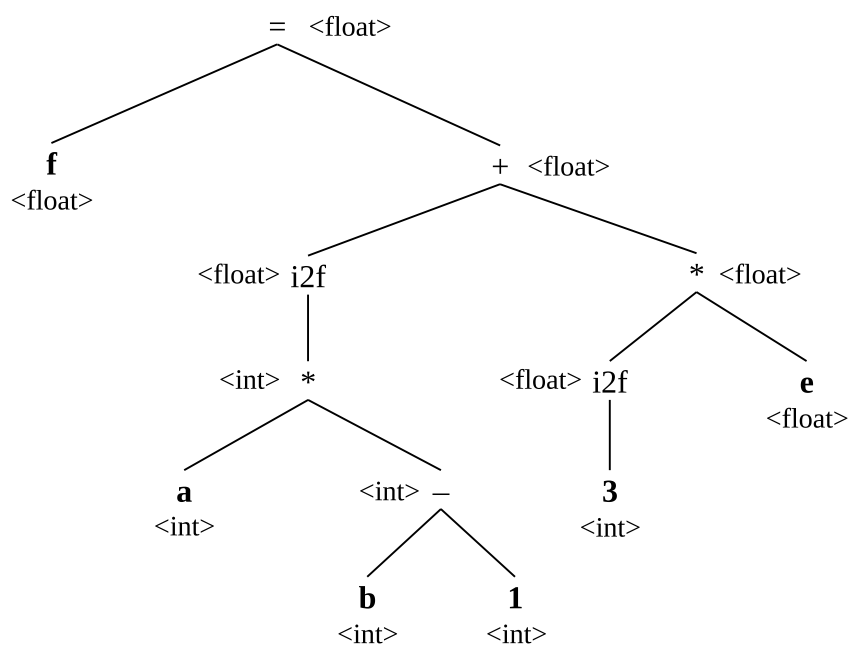
- Strong vs. weak typing
  - strong typing
    - no implicit type conversions are possible
    - Pascal, SnuPL/1
  - weak typing
    - allows for certain implicit type conversions
      - i.e., interpret a string as a number
    - Java, Perl permit a large number of implicit type conversions

      ```
      i = a*2
      ```

      works not only if a is an integer but also when it is NULL, undefined, a string, or an array
  - C is weakly, statically typed

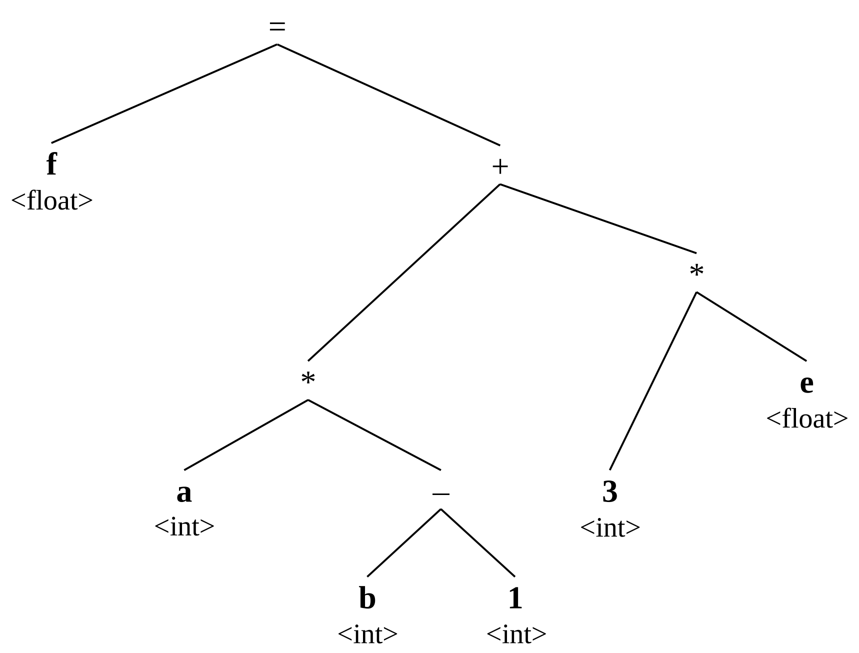
Type Checking Formalism

- Type Checking
  - verifying fully typed programs
    - all nodes in the AST are fully typed, i.e., the process is reduced to verifying that the types of nodes and their children are correct



Type Checking Formalism

- Type Interference
  - deducing missing type information from partially typed programs
    - “filling in missing type information”
    - AST has only partial type information (variables, numbers), deduce types of nodes without type information



Type Checking Formalism

- Inference rules
  - used in type inference to compute the missing types
  - in the form “if *hypothesis* is true, then *conclusion* is true”.

$$\frac{\vdash \text{hypothesis} \;\ldots\; \vdash \text{hypothesis}}{\vdash \text{conclusion}}$$

  - notation
    - x: T x is of type T
    - $\vdash$ infers, “it is provable that”
    - $\wedge$ and
    - $\Rightarrow$ implication

Type Checking Formalism

- Type Inference rules

$$\frac{\vdash e_1: \text{int} \quad \vdash e_2: \text{int}}{\vdash e_1+e_2: \text{int}}$$

"if e_1 is of type **int** and e_2 is of type **int**, then e_1+e_2 is of type **int**"

$$\frac{i \text{ is an integer literal}}{\vdash i: \text{int}}$$

Type Checking Formalism

- Example: prove that -1+5 is of type int

$$\dfrac{\dfrac{\text{-1 is an integer literal}}{\vdash -1:\ \text{int}} \qquad \dfrac{\text{5 is an integer literal}}{\vdash 5:\ \text{int}}}{\vdash -1+5:\ \text{int}}$$

Type Checking Formalism

- Soundness
  - a type system is sound if
    - whenever e: T
    - then e evaluates to a value of type T
  - all type inference rules must be sound

Type Checking Formalism

- Type checking proves facts of the form $e: T$
  - proof is performed on the structure of the AST,
    i.e., the proof has the "shape" of the AST
  - one type inference rule is used for each AST node
  - for a node N computing expression e:
    - the hypotheses are the proofs of types on e's subexpressions
    - the conclusion is the type of e
  - computed in a bottom-up pass over the AST

Type Checking Formalism

- Basic type inference rules

$$\frac{}{\vdash \text{true}: \text{bool}} \qquad \frac{}{\vdash \text{false}: \text{bool}}$$

$$\frac{i \text{ is an integer literal}}{\vdash i: \text{int}} \qquad \frac{s \text{ is a string literal}}{\vdash s: \text{char[]}}$$

$$\frac{v \text{ is a variable}}{\vdash v: ?}$$

Type Checking Formalism

- A type environment defines the types for free variables
  - bound vs. free variables

    a

    $a + b$

    $a := a + b;$

  - the type environment is a function identifier → type

    $O \vdash e: T$

    - read: under the assumption that free variables have the types defined by O, it is provable that e is of type T

  - the information of O is stored in the symbol table

Type Checking Formalism

- The type environment is added to all type inference rules

$$\frac{i \text{ is an integer literal}}{O \vdash i\text{: int}} \qquad \frac{}{O \vdash \textit{false}\text{: bool}}$$

$$\frac{O \vdash e_1\text{: int} \quad O \vdash e_2\text{: int}}{O \vdash e_1 + e_2\text{: int}}$$

  - using O it is simple to defer the type of a variable v

$$\frac{O(v) = T}{O \vdash v\text{: T}}$$

Type Checking in SnuPL/1

- Type checking in SnuPL/1 for scalar types

$$\frac{i \text{ is an integer literal}}{O \vdash i\text{: int}} \qquad \frac{c \text{ is an character literal}}{O \vdash c\text{: char}}$$

$$\frac{}{O \vdash \mathit{false}\text{: bool}} \qquad \frac{}{O \vdash \mathit{true}\text{: bool}}$$

$$\frac{O \vdash e_1\text{: T} \;\; \vdash e_2\text{: T}}{O \vdash e_1 \oplus e_2\text{: T}} \qquad \text{for } \oplus \text{ in } \{ +, -, *, / \} \text{ and T = int}$$

$$\frac{O \vdash e_1\text{: T} \;\; \vdash e_2\text{: T}}{O \vdash e_1 \oplus e_2\text{: T}} \qquad \text{for } \oplus \text{ in } \{ \&\&, \| \} \text{ and T = bool}$$

$$\frac{O \vdash e\text{: int}}{O \vdash +/- \; e\text{: int}} \qquad \frac{O \vdash e\text{: bool}}{O \vdash !e\text{: bool}}$$

Type Checking in SnuPL/1

- Type checking in SnuPL/1 for scalar types

$$\frac{O \vdash e_1 : T \quad \vdash e_2 : T}{O \vdash e_1 \oplus e_2 : \text{bool}} \quad \text{for } \oplus \text{ in } \{ =, \# \} \text{ and } T \text{ in } \{ \text{int}, \text{bool}, \text{char} \}$$

$$\frac{O \vdash e_1 : T \quad \vdash e_2 : T}{O \vdash e_1 \oplus e_2 : T} \quad \text{for } \oplus \text{ in } \{ <, <=, >, >= \} \text{ and } T = \text{int}$$

$$\frac{O \vdash e_1 : T \quad \vdash e_2 : T}{O \vdash e_1 := e_2 : T} \quad \text{for } T \text{ in } \{ \text{int}, \text{bool}, \text{char} \}$$

- Can be implemented during a single pass (bottom-up) to the root