Semantic Analysis

Ronghui Gu
Spring 2019
Columbia University

* Course website: https://www.cs.columbia.edu/~rgu/courses/4115/spring2019
** These slides are borrowed from Prof. Edwards.
Semantic Analysis

int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
Static Semantic Analysis

Lexical analysis: Each token is valid?

```java
if i 3 "This" /* valid Java tokens */
#a1123 /* not a token */
```

Syntactic analysis: Tokens appear in the correct order?

```java
for ( i = 1 ; i < 5 ; i++ ) 3 + "foo"; /* valid Java syntax */
for break /* invalid syntax */
```

Semantic analysis: Names used correctly? Types consistent?

```java
int v = 42 + 13; /* valid in Java (if v is new) */
return f + f(3); /* invalid */
```
What’s Wrong With This?

\[ a + f(b, c) \]

- Is \( a \) defined?
- Is \( f \) defined?
- Are \( b \) and \( c \) defined?
- Is \( f \) a function of two arguments?
- Can you add whatever \( a \) is to whatever \( f \) returns?
- Does \( f \) accept whatever \( b \) and \( c \) are?
What’s Wrong With This?

\[ a + f(b, c) \]

Is \( a \) defined?

Is \( f \) defined?

Are \( b \) and \( c \) defined?

Is \( f \) a function of two arguments?

Can you add whatever \( a \) is to whatever \( f \) returns?

Does \( f \) accept whatever \( b \) and \( c \) are?

Scope questions  Type questions
What To Check

Examples from Java:

Verify names are defined (\textit{scope}) and are of the right type (\textit{type}).

```java
int i = 5;
int a = z;  // Error: cannot find symbol */
int b = i[3];  // Error: array required, but int found */
```

Verify the type of each expression is consistent (\textit{type}).

```java
int j = i + 53;
int k = 3 + "hello";  // Error: incompatible types */
int l = k(42);  // Error: k is not a method */
if ("Hello") return 5;  // Error: incompatible types */
String s = "Hello";
int m = s;  // Error: incompatible types */
```
Scope - What names are visible?
Names | Bindings | Objects
---|---|---
Name1 | Obj 1
Name2 | Obj 2
Name3 | Obj 3
Name4 | Obj 4
Scope

Scope: where/when a name is bound to an object
Useful for modularity: want to keep most things hidden

<table>
<thead>
<tr>
<th>Scoping Policy</th>
<th>Visible Names Depend On</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Textual structure of program</td>
</tr>
<tr>
<td></td>
<td>Names resolved by compile-time symbol tables</td>
</tr>
<tr>
<td></td>
<td>Faster, more common, harder to break programs</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Run-time behavior of program</td>
</tr>
<tr>
<td></td>
<td>Names resolved by run-time symbol tables, e.g., walk the stack looking for names</td>
</tr>
<tr>
<td></td>
<td>Slower, more dynamic</td>
</tr>
</tbody>
</table>
A name begins life where it is declared and ends at the end of its block.

“The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.”
Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

“If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.”

```c
void foo()
{
    int x;
    while ( a < 10 ) {
        int x;
    }
}
```
Basic Static Scope in O’Caml

A name is bound after the “in” clause of a “let.” If the name is re-bound, the binding takes effect after the “in.”

let x = 8 in
let x = x + 1 in

Returns the pair (12, 8):

let x = 8 in
(let x = x + 2 in
 x + 2),
x
The “rec” keyword makes a name visible to its definition. This only makes sense for functions.
## Static vs. Dynamic Scope

<table>
<thead>
<tr>
<th>Language</th>
<th>Code Example</th>
</tr>
</thead>
</table>
| C        | int a = 0;
|          | int foo() {
|          |   return a;
|          | } |
|          | int bar() {
|          |   int a = 10;
|          |   return foo();
|          | } |
| OCaml    | let a = 0 in
|          | let foo x = a in
|          | let bar =
|          |   let a = 10 in
|          |   foo 0
| Bash     | a=0
|          | foo ()
|          | {|
|          |   echo $a
|          | } |
|          | bar ()
|          | {|
|          |   local a=10
|          |   foo
|          | } |
|          | bar
|          | echo $a
Most modern languages use static scoping.
Easier to understand, harder to break programs.
Advantage of dynamic scoping: ability to change environment.
A way to surreptitiously pass additional parameters.
Symbol Tables

• A symbol table is a data structure that tracks the current bindings of identifier
• Scopes are nested: keep tracks of the current/open/closed scopes.
• Implementation: one symbol table for each scope.
Implementing C-style scope (during walk over AST):

```c
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    }
    b + x;
}
```
Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table

```c
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    } }
    b + x;
}
```
Symbol Tables by Example: C-style

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous

```
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    }
    b + x;
}
```

\[ x \mapsto \text{int} \]
Symbol Tables by Example: C-style

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous

```c
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    }
    b + x;
}
```
Symbol Tables by Example: C-style

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous

```c
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    }
    b + x;
}
```

$x \rightarrow \text{int}$

$a \rightarrow \text{int}, b \rightarrow \text{int}$

$b \rightarrow \text{float}$
Symbol Tables by Example: C-style

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous
- Reach an identifier: lookup in chain of tables

```c
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    }
    b + x;
}
```
Symbol Tables by Example: C-style

Implementing C-style scope (during walk over AST):

- Reach a declaration: Add entry to current table
- Enter a “block”: New symbol table; point to previous
- Reach an identifier: lookup in chain of tables
- Leave a block: Local symbol table disappears

```c
int x;
int main() {
    int a = 1;
    int b = 1; {
        float b = 2;
        for (int i = 0; i < b; i++) {
            int b = i;
            ...
        }
    }
    b + x;
}
```

\[ x \mapsto \text{int} \]

\[ a \mapsto \text{int}, b \mapsto \text{int} \]
Types - What operations are allowed?
A restriction on the possible interpretations of a segment of memory or other program construct.

Two uses:

**Safety:** avoids data being treated as something it isn’t

**Optimization:** eliminates certain runtime decisions
Safety - Why do we need types?

Certain operations are legal for certain types.

```c
int a = 1, b = 2;
return a + b;
```

```c
int a[10], b[10];
return a + b;
```
C was designed for efficiency: basic types are whatever is most efficient for the target processor.

On an (32-bit) ARM processor,

```c
char c;        /* 8-bit binary */
short d;       /* 16-bit two’s-complement binary */
unsigned short d; /* 16-bit binary */
int a;         /* 32-bit two’s-complement binary */
unsigned int b; /* 32-bit binary */
float f;       /* 32-bit IEEE 754 floating-point */
double g;      /* 64-bit IEEE 754 floating-point */
```
Misbehaving Floating-Point Numbers

\[ 1e20 + 1e-20 = 1e20 \]
\[ 1e-20 \ll 1e20 \]

\[(1 + 9e-7) + 9e-7 \neq 1 + (9e-7 + 9e-7)\]

\[ 9e-7 \ll 1, \text{ so it is discarded, however, 1.8e-6 is large enough} \]

\[ 1.00001(1.000001 - 1) \neq 1.00001 \cdot 1.000001 - 1.00001 \cdot 1 \]
\[ 1.00001 \cdot 1.000001 = 1.00001100001 \text{ requires too much intermediate precision.} \]
What’s Going On?

Floating-point numbers are represented using an exponent/significand format:

\[
\begin{align*}
1 & \quad \overbrace{10000001} \quad \overbrace{0110000000000000000000000000000} \\
S & \quad \text{8-bit exponent} \quad E \quad \text{23-bit significand} \quad M \\
& = -1^S \times (1.0 + 0.M) \times 2^{E-bias} \\
& = -1.0112 \times 2^{129-127} = -1.375 \times 4 = -5.5.
\end{align*}
\]

What to remember:

\[
\boxed{1363.456846353963456293}
\]

represented \quad rounded
Results are often rounded:

\[
\begin{array}{c}
1.00001000000 \\
\times 1.00000100000 \\
\hline
1.00001100001
\end{array}
\]

(rounded)

When \( b \approx -c \), \( b + c \) is small, so \( ab + ac \neq a(b + c) \) because precision is lost when \( ab \) is calculated.

Moral: Be aware of floating-point number properties when writing complex expressions.
Type Systems
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.

• Three kinds 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 (Python)
  • **Untyped**: No type checking (machine code)
Statically-typed: compiler can determine types.

Dynamically-typed: types determined at run time.

Is Java statically-typed?

```java
class Foo {
    public void x() {
    }
}

class Bar extends Foo {
    public void x() {
    }
}

void baz(Foo f) {
    f.x();
}
```
Strongly-typed Languages

Strongly-typed: no run-time type clashes (detected or not).

C is definitely not strongly-typed:

```c
float g;
union { float f; int i } u;
u.i = 3;
g = u.f + 3.14159; /* u.f is meaningless */
```

Is Java strongly-typed?
• Type Checking is the process of verifying fully typed programs.
• Type Inference is the process of filling in missing type information.
• Inference Rules: formalism for type checking and inference.
Inference Rules

Inference rules have the form If Hypotheses are true, then Conclusion is true

\[ \vdash \text{Hypothesis}_1 \quad \vdash \text{Hypothesis}_2 \]
\[ \vdash \text{Conclusion} \]

Typing rules for int:

\[ \vdash \text{NUMBER} : \text{int} \]

\[ \vdash \text{expr}_1 : \text{int} \quad \vdash \text{expr}_2 : \text{int} \]
\[ \vdash \text{expr}_1 \text{ OPERATOR} \text{ expr}_2 : \text{int} \]

Type checking computes via reasoning
How To Check Expressions: Depth-first AST Walk

check: environment $\rightarrow$ node $\rightarrow$ typedNode

```
1 - 5
  /
  1
/
5
```

check(−)
check(1) = 1 : int
check(5) = 5 : int
int − int = int
= 1 − 5 : int

```
1 + "Hello"
  /
  1
/
"Hello"
```

check(+)
check(1) = 1 : int
check("Hello") = "Hello" : string
FAIL: Can’t add int and string
How To Check Symbols?

What is the type of a variable reference?

\[ \frac{x \text{ is a symbol}}{\vdash x : ?} \]

The local, structural rule does not carry enough information to give \( x \) a type.
Put more information in the rules!

A type environment gives types for free variables.

\[
\begin{align*}
\mathcal{E} & \vdash \text{NUMBER} : \text{int} \\
\mathcal{E}(x) & = \mathbf{T} \\
\mathcal{E} & \vdash x : \mathbf{T} \\
\mathcal{E} & \vdash \text{expr}_1 : \text{int} \quad \mathcal{E} & \vdash \text{expr}_2 : \text{int} \\
\mathcal{E} & \vdash \text{expr}_1 \ \text{OPERATOR} \ \text{expr}_2 : \text{int}
\end{align*}
\]
The environment provides a “symbol table” that holds information about each in-scope symbol.
The Type of Types

Need an OCaml type to represent the type of something in your language.

For MicroC, it’s simple (from ast.ml):

```ocaml
type typ = Int | Bool | Float | Void
```

For a language with integer, structures, arrays, and exceptions:

```ocaml
type ty = (* can’t call it "type" since that’s reserved *)
  Void
  | Int
  | Array of ty * int (* type, size *)
  | Exception of string
  | Struct of string * ((string * ty) array) (* name, fields *)
```
module StringMap = Map.Make(String)

type symbol_table = {
  (* Variables bound in current block *)
  variables : ty StringMap.t
  (* Enclosing scope *)
  parent : symbol_table option;
}