
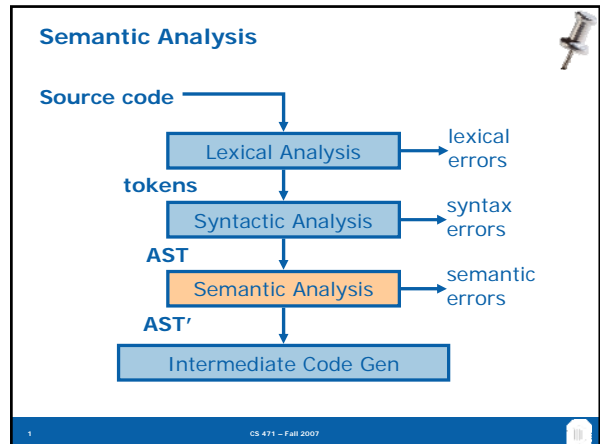


Semantic Analysis: Tiger Type Checking

CS 471
October 15, 2007

Goals of a Semantic Analyzer

Find all possible remaining errors that would make program invalid

Static checks

- Done by the compiler
- Detect and report errors by analyzing the sources

Dynamic checks

- Done at run time
- Detect and handle errors as they occur

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

Program Checking

Why do we care?

Obvious:

- Report mistakes to programmer
- Avoid bugs: *f[6]* will cause a run-time failure
- Help programmer verify intent

How do these checks help compilers?

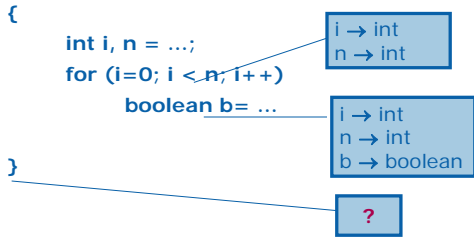
- Allocate right amount of space for variables
- Select right machine operations
- Proper implementation of control structures

Scoping: Variables/Identifiers

Need an environment that keeps track of types of all identifiers in scope

```

{
  int i, n = ...;
  for (i=0; i < n; i++)
    boolean b = ...
}
  
```



Scope Issues

One idea is to have a **global symbol table** and **save the scope information for each entry**.

- When an identifier goes out of scope, scan the table and remove the corresponding entries
- We may even link all same-scope entries together for easier removal

6 CS 471 – Fall 2007

Symbol Tables

purpose:

- keep track of names declared in the program
- names of
 - variables, functions, classes, types, ...

symbol table entry:

- associates a name with a set of attributes, e.g.:
 - kind of name (variable, type, function, etc)
 - type (int, float, etc)
 - nesting level
 - size
 - memory location (i.e., where will it be found at runtime)

7 CS 471 – Fall 2007

Symbol Tables in Tiger

- **Insert** name into a symbol table during declaration processing
- **Lookup** names to process variable references
- **Fill in** attributes: assumes where symbol is stored
- **Enter** scope
- **Exit** scope

8 CS 471 – Fall 2007

Symbol Table Implementation (Tiger)

Two structures: Hash table, Scope Stack

Symbol = foo
Hash(foo) = i

Symbol table

9 CS 471 – Fall 2007

Enter/Exit Scope

We also need a **stack** to keep track of the "nesting level" as we traverse the tree...

10 CS 471 – Fall 2007

Tiger Symbol Tables

Two namespaces ... two symbol tables

- Namespaces for types
- Namespaces for vars and functions
- Called **tenv** and **venv**, respectively

11 CS 471 – Fall 2007

Tenv: Type Environment

[See types.h]

X → types

```

Struct Ty_ty_ {
  enum{Ty_record, Ty_nil, Ty_int, Ty_string, Ty_array,
    Ty_name, Ty_void} kind;
  union {Ty_fieldList record;
    Ty_ty array;
    struct {S_symbol sym; Ty_ty ty;} name;
  } u;
};

```

12 CS 471 – Fall 2007

Type Creation and Query

Initial tenv

Create a new int/string:

```

Ty_int()
Ty_str()

```

Is it an int/string?

```

if (left.type == Ty_int())

```

13 CS 471 – Fall 2007

Type Environment

Recall structural vs. name equivalence

```

let type a = {x:int, y:int}
  type b = {x:int, y:int}
  var i : a := ...
  var j : b := ...
in i := j end

```

```

let type a = {x:int,y:int}
  type c = a
  var i : a := ...
  var j : c := ...
in i := j end

```

14 CS 471 – Fall 2007

Venv

[See env.h]

X → E_venv

```

Struct E_venv_{
  enum{E_varEntry,E_funEntry} kind;
  union{struct {Ty_ty ty;} var;
    struct {Ty_tyList formals; Ty_ty result;} fun;
  } u;
};

```

```

E_venv E_VarEntry(Ty_ty ty);
E_venv E_FunEntry(Ty_tyList formals, Ty_ty result);
S_table E_base_tenv(void);
S_table E_base_venv(void);

```

15 CS 471 – Fall 2007

Initial venv

Base functions from Appendix A

16 CS 471 – Fall 2007

High-Level View of the Project

```

types.c
env.c
semant.c
  - Type checking
  - Generate intermediate code

```

For now (PA5), semant.c will just do type checking

```

- int code = NULL pointer

```

Later, semant.c will generate intermediate code

17 CS 471 – Fall 2007

We Will Write

Recursive functions to operate on the AST

```
transTy:
  A_type * tenv → Ty_ty
transVar:
  A_var * tenv * venv → Ty_ty (+ int code)
transExp:
  A_var * tenv * venv → Ty_ty (+ int code)
transDec:
  A_var * tenv * venv → () /* side effect symtab */
```

18

CS 471 – Fall 2007

transTy – for name types

Given symbol **n**:

1. Lookup **n** in **tenv**
2. Return **Ty_Name**
(or return result of lookup. If lookup fails return an error and assume int)



19

CS 471 – Fall 2007

transTy – for record types

Example

```
y: { a:x;
    b:y;
    c:z; }
```

} Field list

a	type x	
b	type y	
c	type z	

1. Lookup **x, y, z** in **tenv**
2. Place the tyfields list in a record type and return it

20

CS 471 – Fall 2007

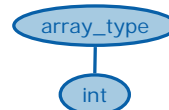
transTy – for array types

type **t** = array of **tp**

1. Lookup type of **tp** in **tenv**
2. Return an array type

Example:

type **t** = array of int



21

CS 471 – Fall 2007

For Recursive Types

```
type t1 = record { x: int; y: t2; }
type t2 = record { a: string; b: t1; }
```

1. Put all "headers" in the environment first
header: type **t1** =
S_enter(tenv, t1, Ty_Name(t1, NULL))
2. Process all of the "bodies"
body: record {x: int; y:t2}

22

CS 471 – Fall 2007

We Will Write (revisited)

```
transTy:
  A_type * tenv → Ty_ty
transVar:
  A_var * tenv * venv → Ty_ty (+ int code)
transExp:
  A_var * tenv * venv → Ty_ty (+ int code)
transDec:
  A_var * tenv * venv → () /* side effect symtab */
```

23

CS 471 – Fall 2007

Summary

Type checking is simple a walk of our AST

Upcoming Events:

- * PA5: Type Checking (October 26)

Notes:

- * No class on Monday, November 19

