CS 473: COMPILER DESIGN
• Goal: take the sequence of tokens produced by lexing, and turn them into an AST, to make the structure of the program clear

• Method:
  1. Write a file (.y, .grm) that describes the grammar and disambiguates it, and explains how to build the AST node for each production
  2. Use a parser generator to turn that grammar into a shift-reduce automaton, encoded in an action-goto table

• Lexer generators and parser generators work together to define the syntax of a language
• Lexing and parsing together turn program text into a representation that starts to suggest meaning/behavior of programs

• Next up: looking for meaning
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0; }

Abstract Syntax Tree:
```
If
  Eq
  Assn
  b
  0
  a
  1
```

Assembly Code
```
11:
  cmpq %eax, $0
  jeq 12
  jmp 13
12:
  ...
```
Syntax and Semantics

• Lexing and parsing deal with questions of *syntax*:
  – What symbols are allowed to appear in programs? What are the pieces of programs? What do programs look like?

• The rest of the compiler looks at the *semantics* of programs:
  – What do programs mean? What does this operator/command do? What is the output of this supposed to be?

• We start by gathering information from the program structure, and looking for errors in that information: *semantic analysis*

  Syntax errors: Semantic errors:

  int b 4;
  y x <;
  if w x y z;

  int b = “hi”;
  a = b + 1; // b undeclared
Scope and Types

- In most programming languages, the main operation is assigning values to variables
  - In all languages, we need to know what variables are in scope
  - In some languages, we need to know the types of variables

- We store this information in *environments* (alternatively, *symbol tables*)

```cpp
int a = 3;
string b = “Hi”;

{a : int, b : string}
```

"binding for b"

<table>
<thead>
<tr>
<th>a</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>string</td>
</tr>
</tbody>
</table>
Scope and Types

• In most programming languages, the main operation is assigning values to variables
  – In all languages, we need to know what variables are in scope
  – In some languages, we need to know the types of variables

• We store this information in environments (alternatively, symbol tables)

```plaintext
int a = 3;
string b = “Hi”;

{a : int, b : string}
```

<table>
<thead>
<tr>
<th>a</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>string</td>
</tr>
</tbody>
</table>
• In most programming languages, the main operation is assigning values to variables
  – In all languages, we need to know what variables are in scope
  – In some languages, we need to know the types of variables

• We store this information in environments (alternatively, symbol tables)

```c
int a;
string b;

if(a == 1){
    char a = ‘a’;
    printf(“%d\n”, a);
}
```
Scope and Types

- In most programming languages, the main operation is assigning values to variables
  - In all languages, we need to know what variables are in scope
  - In some languages, we need to know the types of variables

- We store this information in environments (alternatively, symbol tables)

```c
int a;
string b; // {a : int, b : string}

if(a == 1){
    char a = ‘a’; // {a : char, b : string}
    printf(“%d
", a);
}
// {a : int, b : string}
```
• In most programming languages, the main operation is assigning values to variables
  – In all languages, we need to know what variables are in scope
  – In some languages, we need to know the types of variables

• We store this information in environments (alternatively, symbol tables)

```c
int a;
string b;
if(a == 1){
    char a = 'a';
    printf("%d\n", a);
}
```
Scope and Types

- In most programming languages, the main operation is assigning values to variables
  - In all languages, we need to know what variables are in *scope*
  - In some languages, we need to know the *types* of variables

- We store this information in *environments* (alternatively, *symbol tables*

```c
int a;
string b;
if(a == 1){
    char a = 'a';
    printf("%d\n", a);
}
```
• In most programming languages, the main operation is assigning values to variables
  – In all languages, we need to know what variables are in scope
  – In some languages, we need to know the types of variables

• We store this information in environments (alternatively, symbol tables)

```c
int a;
string b;
if(a == 1){
    char a = ‘a’;
    printf(“%d\n”, a);
}
```
Scope and Types

• In most programming languages, the main operation is assigning values to variables
  – In all languages, we need to know what variables are in scope
  – In some languages, we need to know the types of variables

• We store this information in environments (alternatively, symbol tables)

Imperative table: have a stack of bindings for each variable
  push when defined,
  pop when scope ends

Functional table: keep old table around, switch back when scope ends

```
Prog
  Decl
    a : int, b : string
  If
    a : char, b : string
  ... 

Body
  Decl
    a
  int
  a
  a
```

12
Kinds of Bindings

• A language may have multiple *namespaces*
  – Examples: variables, function names, global v. local, type names, class names

• Each namespace gets its own environment/symbol table:

```plaintext
type mine = array of int
var mine : int := 1

Types: { mine = array of int }
Vars/Funs: { mine : int }

var mine : mine
```
Kinds of Bindings

• A language may have multiple *namespaces*
  – Examples: variables, function names, global v. local, type names, class names

• Each namespace gets its own environment/symbol table:

```plaintext
type mine = array of int
var mine : int := 1
function mine(int x) : int = x + 1

Types: { mine = array of int }
Vars/Funs: { mine : int }

Type mine = array of int

Types: { mine = array of int }
Vars/Funs: { mine : function(int, int) }
```

• In C, there are a few different namespaces, but no symbol can appear in more than one
Top
Type Checking

- Goal: catch programs that we know won’t run properly, before we bother translating them

```java
int a = "Hi";
```

- Avoid wasting time and effort translating something that won’t mean anything
- Avoid translating into machine code that *does* mean something, but the wrong thing

```
a = 0x123456
```

- Approach: recurse through the AST, building up environments and computing types of expressions
Type Checking

• Goal: catch programs that we know won’t run properly, before we bother translating them
• Recurse through the AST, building up environments and computing types of expressions

```
Vars: {a : int, ...}
```

```
Plus

1

a

type: int

1

a

type: int
type: int
```
Type Checking

• Goal: catch programs that we know won’t run properly, before we bother translating them
• Recurse through the AST, building up environments and computing types of expressions

Vars: \{a : \text{string}, \ldots\}

\[
\begin{array}{c}
\text{Plus} \\
1 \quad a
\end{array}
\]

\text{type: int} \quad \text{type: string}

type error!

• Gets more complicated with more complicated types!
  – Functions, arrays, records/structs/objects, recursive types, …
Type Checking: Kinds of Types in Tiger

- **Primitive types**: int, string
  
  
  
  123 "Hi!" var a : int := 3

- **Arrays**

  type arrtype = array of int

  var arr1 : arrtype := arrtype [10] of 0

  arr1[3]

- **Records**

  type rectype = {name : string, age : int}

  var rec1 : rectype := rectype {name = "Nobody", age = 1000}

  rec1.name := "Somebody"

- **Void** (for functions that don’t return a value)
Type Checking: Variables

```
type result = int
var a : result := 1
```

```
Vars: {a : int, ...}
```

```
Plus

1

a

type: int
```

```
Vars: {a : result, ...}
```

```
a

type: result
```

Type Checking: Variables

```
type result = int
var a : result := 1
```

```
Plus

1

type: int

a

type: int

Vars: {a : int, ...}

type result = int

Vars: {a : result, ...}

Types: {result = int, ...}
```
var b := f(a, 1)

Vars/Funs: \{f : ([\text{string}, \text{int}], \text{int}), \ldots\}\n
Vars/Funs: \{f : \text{int} f (\text{string}, \text{int}), \ldots\}\n
Vars/Funs: \{f : \text{string} \times \text{int} \to \text{int}\}
Type Checking: Functions

var b := f(a, 1)

vars/funs: {f : ([string, int], int), ...}

call

f

a

l

type: ([string, int], type: string type: int int)

type: int

type: (int)

type: int

• Look up function signature in the environment, check arguments, result is return type
Type Checking: Arrays and Records

var b := a[1]

- Array access acts like an operator, taking an (array of T) and an int and returning a T

var b = a.name

- For field access, field name is an index into the record type of the LHS
Questions

Top
Type Checking: Declarations

- The bindings in environments come from *declarations*
- In some languages (including Tiger and ANSI C), declarations appear in a block at the start of a section of code; in others, they can appear anywhere

```plaintext
let

var a := 5  
int a = 5;

type t := int  
typedef int t;

function f(int x) := ...
int f(int x){ ...

in

<code that uses a, t, f>

int main(){
<code that uses a, t, f>
}
```
Type Checking: Declarations

- The bindings in environments come from *declarations*
- In some languages (including Tiger and ANSI C), declarations appear in a block at the start of a section of code; in others, they can appear anywhere

```plaintext
let

  var a := 5
  type t := int          typedef int t;
  function f(int x) := ...  int f(int x){ ... }  

in

  int main(){
    <code that uses a, t, f>
    int a = 5;
    <code that uses a, t, f>
  }
```
Program 2 Questions

Top
Type Checking: Declarations

- The bindings in environments come from *declarations*
- In some languages (including Tiger and ANSI C), declarations appear in a block at the start of a section of code; in others, they can appear anywhere

```plaintext
let
  var a := 5
  type t := int              typedef int t;
  function f(int x) := ...
                          int f(int x){ ... }
in
  <code that uses a, t, f>       <code that uses t, f>
  int a = 5;
  <code that uses a, t, f>
}
```
Type Checking: Declarations

- Variable declarations:

\[
\text{var } a : \text{int} := 5 \\
\text{var } a := 5
\]

Vars: \{b : \text{string}\} a : \text{int}\}
Variable declarations:

```
var a : int := 5
var a := 5
```

Types: `{x = string} t = int`

Type declarations:

```
type t = int
```

Types: `{x = string} t = int`
Type Checking: Declarations

- Function declarations:

function f(x : int, y : int) : int := x + y

Vars/Funs: {b : string, a : int}

- First, typecheck the body
Function declarations:

function f(x : int, y : int) : int := x + y

First, typecheck the body
  — in an environment that includes the parameters!

Then, add the function signature to the top-level environment
Questions
Type Checking: Recursive Declarations

• Function declarations:

    function f(x : int, y : int) : int := if y = 0 then 1 else x + f(y – 1)

    Vars/Funs: {b : string, a : int}

• First, typecheck the body
  – in an environment that includes the parameters!
Function declarations:

function f(x : int, y : int) : int := if y = 0 then 1 else x + f(y - 1)

First, typecheck the body
  – in an environment that includes the parameters!
• Function declarations:

function f(x : int, y : int) : int := if y = 0 then 1 else x + f(y – 1)

\{f : ([int, int], int)\}

Vars/Funs: \{b : string, a : int\}

\ f : ([int, int], int)\}

• First, typecheck the body
  – in an environment that includes the parameters, \textit{and} the function signature!
  – We take the function for granted while type-checking the function itself!

• Then, add the function signature to the top-level environment
Semantic Analysis: Summary

- Goal: gather information about elements of the program, and catch any errors that were too tricky for parsing to catch

- Method: recurse through the AST, building up information (stored in environments/symbol tables) and checking for errors

- Language design questions: What kinds of names conflict? What kinds can coexist (e.g., type a vs. var a)? What types are in the language? What do we do when we see an error?

- Note: type checking can get much more complicated, with generics, classes, first-class functions, ...
  - See also CS 476: Programming Language Design

- Next up: Now that we know what variables there are, prepare to store them