Announcements

- Exam 1
- I will update Rainbow table after class: HW1-2, Quiz 1-2
- Next class: HW1-3, Quiz1-3, Exam 1

- HW4 will be up after class
  - Scoping and semantics (attribute grammars)
  - You should be able to complete 100% of homework after class today

Last Class

- Naming, binding and scoping
  - Notion of binding time
  - Object lifetime and storage management
  - Scoping

- Today: A brief recap of scoping

Scoping, Recap

- In most programming languages a variable name can be used multiple times

  - Scoping rules: map variable names to declarations
  - Scope: region of program text where declaration is visible (region of program where a binding is active)

- Most languages use static scoping
  - Mapping from uses to declaration is made at compile time

Static Scoping

- Also known as lexical scoping
  - Block structure and nesting of blocks gives rise to the closest nested scope rule
    - There are local variable declaration within a block
    - A block inherits variable declarations from enclosing blocks
    - Local declarations take precedence over inherited ones
      - Hole in scope of inherited declaration
      - In other words, inherited declaration is hidden
  - Lookup for non-local variables proceeds from inner to outer enclosing blocks

Example with Frames

- Rule: a variable is visible if it is declared in its own block or in a textually surrounding block and is not ‘hidden’ by a binding to it in a closer block (i.e., hole in scope)
Dynamic Scoping

- Allows for local variable declaration
- Inherit non-local variables from subroutines that are live when current subroutine is invoked
- Use of variable is mapped to the declaration of that variable in the most recently invoked and not yet terminated stack frame

An Important Note!

- For now, we assume languages that do not allow subroutines to be passed as arguments or returned from other subroutines, i.e., subroutines (functions) are third-class values
  - When subroutines (functions) are third-class values, it is guaranteed the static reference environment is on the stack
  - i.e., a subroutine cannot outlive its reference environment

- Static scoping rules become more involved in languages that allow subroutines to be passed as arguments and returned from other subroutines, i.e., subroutines (functions) are first class values

- We will return to scoping later during our discussion of functional programming languages

Static vs. Dynamic Scoping

- Static Scoping:
  - a bound to R.a
  - b to main.b
  - c to main.c

- Dynamic Scoping:
  - a bound to R.a
  - b to main.b
  - c to P.c
Semantic Analysis

Reading: Scott, Chapter 4.1-4.3

Today’s Lecture Outline

- Static Semantic Analysis
  - Syntax vs. static semantics
  - Static semantics vs. dynamic semantics
  - Attribute Grammars
    - Attributes and rules
- Next class:
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars

Static Semantics

- Earlier we considered syntax analysis
  - Informally, syntax deals with the form of programming language constructs
- We now look at static semantic analysis
  - Semantics deals with the meaning of programming language constructs
- The distinction between the two is fuzzy
  - In practice, anything that is not expressed in terms of certain CFG (LALR(1), in particular) is considered semantics

Static Semantics vs. Dynamic Semantics

- Static semantic analysis (compile-time)
  - Informally, reasons about properties statically, before program execution
    - E.g., determine static types of expressions, detect certain errors
- Dynamic semantic analysis (run-time)
  - Reasons about properties dynamically, during program execution
    - E.g., could expression a[i] index out of array bounds, etc.?

The Role of Semantic Analysis

- Detect errors in programs!
- Static semantic analysis
  - Detect as many errors as possible early, before execution
    - Type inference and type checking
- Dynamic semantic analysis
  - Detect errors by performing checks during execution
    - Tradeoff: dynamic checks slow program execution
- Languages differ greatly in the amount of static semantic analysis and dynamic semantic analysis they perform

Question

- Let’s look at 3 languages we are all familiar with: C++, Java and Python
- How do these languages differ in the static semantic analysis and dynamic semantic analysis they perform?
Examples of Static Semantic Errors

- **Type mismatch:**
  - \( x = y + z + w \): type of left-hand-side does not “match” type of right-hand-side
  - \( A a; \ldots ; a.m() : m() \) cannot be invoked on a variable of type \( A \)

- Definite assignment check in Java: a local variable must be assigned before it is used.

Examples of Dynamic Semantic Errors

- **Null pointer dereference:**
  - \( a.m() \) in Java, and \( a \) is null (i.e., uninitialized reference)
  - What happens?

- Array-index-out-of-bounds:
  - \( a[i] \), \( i \) goes beyond the bounds of \( a \)
  - What happens in C++? What happens in Java?

- Casting an object to a type of which it is not an instance
  - C++? Java?
  - And more…

Dynamic Semantic Errors (C-style casts in C++)

```c++
//#1
void* x = (void*) new B;
B* q = (B*) x; // a safe downcast?
int case1 = q->foo();

//#2
void* x = (void*) new A;
B* q = (B*) x; // a safe downcast?
int case2 = q->foo(); // what happens?
```

Dynamic Semantic Errors (Java)

```java
//#1
Object x = new B();
B q = (B) x; // a safe downcast
int case1 = q.foo();

//#2
Object x = new A();
B q = (B) x;
int case2 = q.foo();
```

Static Semantics vs. Dynamic Semantics

- Again, distinction between the two is fuzzy
- For some programs, the compiler can predict run-time behavior by using **static analysis**
  - E.g., #1: \( q \) always points to a B object, thus \( q.foo() \) calls B’s \( foo() \), not A’s \( foo() \)
  - E.g., #2: \( x \) always points to an A object, thus the compiler can predict the problematic cast
- For arbitrary programs, compiler cannot predict run-time behavior!

Semantic Analyzer

- Scanner, token stream, parser, parse trees, semantic analyzer, and intermediate code generator
- Code generator, assembly code
- Compiler
- Optimizer modifies intermediate form
- Intermediate form
- Semantic analyzer performs static analysis on parse trees and ASTs
- Optimizer performs static analysis on intermediate 3-address code.
Chapter: Attribute Grammars: Foundation for Static Semantic Analysis

- **Attribute Grammars**: generalization of Context-Free Grammars
  - Associate meaning with parse trees
  - **Attributes**
    - Each grammar symbol has one or more values called attributes associated with it
  - **Semantic rules**
    - Each grammar production has associated “rule” which may refer to and compute the values of attributes

**Example**: Attribute Grammar to Compute Value of Expression (denote grammar by AG1)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E1+T</td>
<td>E.val := E1.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val := T.val</td>
</tr>
<tr>
<td>T → T1*F</td>
<td>T.val := T1.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val := F.val</td>
</tr>
<tr>
<td>F → num</td>
<td>F.val := num.val</td>
</tr>
</tbody>
</table>

**Example**: Decorated parse tree for input 3 * 5 + 2 * 4

**Exercise**

- **LR grammar for arithmetic expressions**:

  1. S → E
  2. E → E + T
  3. E → T
  4. T → T * num
  5. T → num

  Write an attribute grammar that translates an expression into prefix notation. The grammar associates an attribute `val` with `S`, which holds the string in prefix. For example, for `5 + 6 * 7`, `val` of `S` will be "+ 5 * 6 7".

  Use *+ to denote string concatenation

**Building an Abstract Syntax Tree (AST)**

- **So far, we talked about parse trees**
- **In fact, compilers use abstract syntax trees**
  - Suitable intermediate representation
  - Define semantic analyses over ASTs
  - AST is basis for translation into intermediate code
- **An AST is an abbreviated parse tree**
  - Operators and keywords do not appear as leaves, but at the interior node that would have been their parent
  - Chains of single productions are collapsed
Building ASTs for Expressions

Parse tree for \(3 \times 5 + 2 \times 4\)

Abstract syntax tree (AST)

```
  +
 /  \
F *  E
  |
F   num:5
  |
F   num:2
```

How do we construct syntax trees for expressions?

Attribute Grammar to build AST for Expression (denote by AG2)

- An attribute grammar:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E \rightarrow E_T)</td>
<td>(E.nptr := mknod(e \times \text{num}, T.nptr))</td>
</tr>
<tr>
<td>(E \rightarrow T)</td>
<td>(E.nptr := T.nptr)</td>
</tr>
<tr>
<td>(T \rightarrow T_F)</td>
<td>(T.nptr := mknod(e \times \text{num}, T_nptr, F.nptr))</td>
</tr>
<tr>
<td>(T \rightarrow F)</td>
<td>(T.nptr := F.nptr)</td>
</tr>
<tr>
<td>(F \rightarrow \text{num})</td>
<td>(F.nptr := mknod(e \times \text{num}, \text{num}.val))</td>
</tr>
</tbody>
</table>

Function \(mknod(op, left, right)\) creates an operator node in the AST with label \(op\), and two fields containing pointers \(left\) and \(right\) .

Function \(mkleaf\) creates a leaf node with label \(\text{num}\), and a field containing the value of the number.

Constructing ASTs for Expressions

Input: \(3 \times 5 + 2 \times 4\)

```
  +
 /  \
F *  E
  |
F   num:5
  |
F   num:2
```

Another Example

Another LR grammar: Input: \(3/5 - 2/4\).

```
E \rightarrow E_T
E \rightarrow T
T \rightarrow T_F
T \rightarrow F
F \rightarrow \text{num}
```

Exercise

- We know that the language \(L = \{a^n b^n c^n\}\) is not context free. It can be captured however with an attribute grammar. Give an underlying CFG and a set of attribute rules that associate an attribute \(\text{ok}\) with the root \(S\) of each parse tree, such that \(S.\text{ok}\) is true if and only if the string corresponding to the fringe of the tree is in \(L\).

Question

- Consider CFG and attribute grammar:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S \rightarrow E)</td>
<td>(\text{print}(E.val))</td>
</tr>
<tr>
<td>(E \rightarrow E_T)</td>
<td>(E.val := E_1.val - T.val)</td>
</tr>
<tr>
<td>(E \rightarrow T)</td>
<td>(E.val := T.val)</td>
</tr>
<tr>
<td>(T \rightarrow \text{num})</td>
<td>(T.val := \text{num}.val)</td>
</tr>
</tbody>
</table>

- What is the result for \(5/3 - 2/4\)?

  Answer: 0
Another Grammar

Now, the LL(1) version of same grammar:

- **E** → **T** **T**
  - **T** → **T** **T** | ε
  - **T** → ε

**T** stands for term
**TT** stands for term_tail

- Goal: construct an attribute grammar that computes the value of an expression
  - Values must be computed “normally”, i.e.,
    - 5-3-2 is 0 not 4

Exercise

- Given a context-free grammar for boolean expressions in preorder
  
  - **S** → **B**
  - **B** → **and** **B** **B**
  - **B** → **or** **B** **B**
  - **B** → **not** **B**
  - **B** → **true**
  - **B** → **false**

construct an attribute grammar that evaluates the boolean expression