Announcements

- Exam 1 is graded. Curved +10
- Rainbow table will be available after class
  - HW1 and HW2,
  - Exam 1,
  - Quizzes 1-3
- HW4 will be up after class

Last Class

- Naming, binding and scoping
  - Notion of binding time
  - Object lifetime and storage management
  - Scoping
- Today: Catch-up

Scoping, Catch-up

- In most 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
  - Block structured PLs (referred as Algol-style structure)
    - Nested subroutines (Pascal, ML, Scheme, etc.)
    - Nested blocks (C, C++ {...} )

Static Scoping in Block Structured Programming Languages

- 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

```
main
a, b, c: integer;
procedure P(){
  c: integer;
  procedure S(){
    c, d: integer;
    procedure R(){
      ...
    } //end R
    R();
  } //end S
  R();
} //end P
procedure R(){
a: integer;
= a, b, c;
} //end R
...
P();
...
} //end main
```

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)

```
main
 --- fp - currently active frame
 --- a
 --- b
 --- c
 at /*1*/
 top
 sp
```

```
Example with Frames

main
a, b, c: integer;
procedure P(){
  c: integer;
  procedure S(){
    c, d: integer;
    procedure R(){
      ...
    } //end R
    R();
  } //end S
  R();
} //end P
procedure R(){
a: integer;
= a, b, c;
} //end R
...
P();
...
} //end main
```

```
Example with Frames

main
a, b, c: integer;
procedure P(){
  c: integer;
  procedure S(){
    c, d: integer;
    procedure R(){
      ...
    } //end R
    R();
  } //end S
  R();
} //end P
procedure R(){
a: integer;
= a, b, c;
} //end R
...
P();
...
} //end main
```

```
Example with Frames

main
a, b, c: integer;
procedure P(){
  c: integer;
  procedure S(){
    c, d: integer;
    procedure R(){
      ...
    } //end R
    R();
  } //end S
  R();
} //end P
procedure R(){
a: integer;
= a, b, c;
} //end R
...
P();
...
} //end main
```

```
Example with Frames

main
a, b, c: integer;
procedure P(){
  c: integer;
  procedure S(){
    c, d: integer;
    procedure R(){
      ...
    } //end R
    R();
  } //end S
  R();
} //end P
procedure R(){
a: integer;
= a, b, c;
} //end R
...
P();
...
} //end main
```
Example

main
---
---
a
b
c
main.P

\[ \text{static link (access link; static environment)} \]

fp

sp
top

at /*3*/

Example

main
---
---
a
b
c
main.R

\[ \text{dynamic link (control link; called-by chain)} \]

dynami

static

link

at /*4*/, P calls main

A Note

- We assume languages that do not allow subroutines to be passed as arguments or returned from other subroutines
- Scoping rules become more involved in languages that allow subroutines to be passed as arguments and/or returned from other subroutines
- We will return to scoping later

Dynamic Scoping

- Allows for local variable declaration
- Inherit non-local variables from subroutines which 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

Dynamic Scoping

- Lookup for non-local variables proceeds from closest dynamic predecessor to the farthest
- Incurs a runtime cost of the lookup
- Used in APL, (old) Lisp, Snobol, Perl

Example

main
---
---
a
b
c
main.P

\[ \text{Which a is modified at /*1*/ under dynamic scoping? Z.a or W.a or both?} \]

main

\[ \text{procedure Z()} \]

\[ a : \text{integer}; \]

\[ a := 1; \]

\[ Y(); \]

\[ \text{output a; } \]

\[ )/\text{end Z;} \]

\[ \text{procedure W()} \]

\[ a : \text{integer}; \]

\[ a := 2; \]

\[ Y(); \]

\[ \text{output a; } \]

\[ )/\text{end W;} \]

\[ \text{procedure Y()} \]

\[ a := 0; /*1*/ \]

\[ )/\text{end Y;} \]

\[ Z(); \]

\[ W(); \]

\[ )/\text{end main;} \]
Example

main calls Z, Z calls Y, Y sets Z.a to 0.

main{
  procedure Z()
  a: integer;
  a := 1;
  Y();
  output a;
  } //end Z;

  procedure W()
  a: integer;
  a := 2;
  Y();
  output a;
  } //end W;

  procedure Y()
  a := 0; /*1*/
  } //end Y;

Z();
W();
} //end main

Example

main calls W, W calls Y, Y sets W.a to 0.

main{
  procedure Z()
  a: integer;
  a := 1;
  Y();
  output a;
  } //end Z;

  procedure W()
  a: integer;
  a := 2;
  Y();
  output a;
  } //end W;

  procedure Y()
  a := 0; /*1*/
  } //end Y;

Z();
W();
} //end main

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

Dynamic Scoping is considered to be a very bad idea. Why?

Today’s Lecture Outline

Static Semantic Analysis
- Syntax vs. Static Semantics
- Static Semantic Analysis vs. Dynamic Semantic Analysis

Attribute Grammars
- Synthesized and inherited attributes
- S-attributed grammars
- L-attributed grammars
Static Semantics

- Earlier we considered syntax analysis
  - Informally, syntax deals with the form of PL constructs
- We now look at static semantic analysis
  - Semantics deals with the meaning of PL 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 that can be determined statically, without program execution
    - E.g., determine static types of expressions, detect certain errors
- Dynamic semantic analysis (run-time)
  - Reasons about properties that depend on program execution
    - E.g., could \( z = x / y \) cause division by zero, could expression \( a[1] \) index out of array bounds, etc.?

The Role of Semantic Analysis

- Detect errors
- 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 all are 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 variable assignment in Java: a local variable must be assigned before it is used

Examples of Dynamic Semantic Errors

- Array-index-out-of-bounds:
  - \( a[i] \): \( i \) goes beyond the bounds of \( a \)
  - What happens in C++? What happens in Java?
- Null pointer dereference:
  - \( a.m() \) in Java or \( a->m() \) in C++, and \( a \) is null
  - 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?

//#3
void* x = (void*) new A;
B* q = (B*)x; //a safe downcast?
int case3 = q->foo(66); //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();

//#3
Object x = new A();
B q = (B)x;
int case3 = q.foo(66);
```

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, the compiler cannot predict run-time behavior!

Static Semantic Analysis

- Optimization
  - E.g., prove that a is never null in a.m() and remove the null-pointer dereference run-time check
  - E.g., prove that a always refers to a subclass of A in x = (A) a and eliminate class-cast check
  - E.g., prove that index i is never out of bounds in a[i] and eliminate array-out-of-bounds check
  - E.g., prove that a always refers to a B object and resolve virtual call a.m() statically
- Static debugging tools
  - E.g., find an execution path where a is null at dereference point a.m()
  - Debugging memory management in C++ programs

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>$S \rightarrow E$</td>
<td>print($E$.val)</td>
</tr>
<tr>
<td>$E \rightarrow E + T$</td>
<td>$E$.val := $E$.val + $T$.val</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E$.val := $T$.val</td>
</tr>
<tr>
<td>$T \rightarrow T * F$</td>
<td>$T$.val := $T$.val * $F$.val</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T$.val := $F$.val</td>
</tr>
<tr>
<td>$F \rightarrow num$</td>
<td>$F$.val := num.val</td>
</tr>
</tbody>
</table>

Example

- **val**: Attributes associated to symbols
  - Intuitively, $A$.val holds the value of the expression, represented by the subtree rooted at symbol $A$
  - Fresh attributes are associated with every node in the parse tree
  - Indices used to distinguish between symbols with same name within same production
    - E.g., $E \rightarrow E_1 + T$ $E$.val := $E_1$.val + $T$.val
  - Attributes of terminals supplied by scanner
  - Attributes of symbols + and * are never used

Exercise

- **LR grammar for arithmetic expressions**:

  $S \rightarrow E$
  $E \rightarrow E + T$
  $E \rightarrow T$
  $T \rightarrow T * F$
  $T \rightarrow F$
  $F \rightarrow num$

- 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
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>$E \rightarrow E_1 + T$</td>
<td>$E.nptr := \text{mknode}(\text{\textquotesingle} + \text{\textquotesingle}, E_1.nptr, T.nptr)$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.nptr := T.nptr$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.nptr := \text{mknode}(\text{\textquotesingle} \ast \text{\textquotesingle}, T_1.nptr, F.nptr)$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.nptr := F.nptr$</td>
</tr>
<tr>
<td>$F \rightarrow \text{num}$</td>
<td>$F.nptr := mkleaf(\text{\textquotesingle}\text{num}\text{\textquotesingle}, \text{num}.val)$</td>
</tr>
</tbody>
</table>

Function $\text{mknode}(op, left, right)$ creates an operator node with label $op$, and two fields containing pointers $left$ and $right$.

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

Another Example

Another LR grammar:

Input: $3/5 - 2/4$

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 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 $L$. 

Exercise