Announcements

- HW2 will be graded by Wednesday morning

Exam Topics

- Regular Expressions
- CFGs
  - Derivation, parsing, ambiguity, operator precedence and associativity
  - LL(1) grammars and parsing
    - FIRST and FOLLOW sets, LL(1) parsing table
    - Obstacles to LL(1)-ness
  - SLR(1) grammars and parsing
    - CFSM and SLR(1) parsing tables
    - Conflicts in SLR(1)

Exam Topics

- Prolog
  - Concepts: Search trees, Unification, Rule ordering, Backtracking, Backward chaining
  - One type of question. Given a Prolog predicate and a query, “Find the first answer” or “Find all answers”
  - Another type of question. Given a description, write a predicate. 3-4 lines of code.

Questions?

- Project?
- Exam?

Last Class

- Concluded Prolog
- Backtracking cut
- Negation by failure
- Generate and test paradigm
  \[
  \text{solve}(...) \leftarrow \text{generate}(...) \text{, test}(S).
  \]
- Prolog in perspective?
Names, Scopes, and Binding

Read: Scott, Chapter 3.1, 3.2 and 3.3.1, 3.3.2 and 3.3.6

Names and Binding

- **Name** - character string which represents a programming language construct (e.g., variable, subroutine)
- **Binding** - an association of a name to the object it represents
  - Can happen at different times during translation and execution

Notion of Binding Time

- **Static**
  - Compile time - often layout of statically defined data in memory is chosen at this time
  - Link time - separately compiled modules of a program are joined together by linker (e.g., adding in standard library routines for I/O)
- **Dynamic**
  - Run time - when program executes

Examples of Binding Time Decisions

- Binding a variable name to memory location
  - Static or Dynamic
  - Determined by scoping rules
- Binding a variable to type
  - Static or Dynamic
- Binding of a call to a target subroutine
  - Static (as it is in C, mostly)
  - Dynamic (virtual calls in Java, C++)

Binding Time and Programming Language Design Choices

- Many PL design choices relate to binding time
- Static binding (also referred to as compile-time binding, early binding)
  - More efficient
  - Less flexible
- Dynamic binding (also referred to as run-time binding, late binding)
  - Less efficient
  - More flexible
Example: Binding Variables to Locations

- Map variable to location
  - Map variable at use to variable declaration
  - Map subroutine at use to target subroutine
- Determined by scoping rules
  - Static scoping (binding before execution)
  - Dynamic scoping (binding during execution)
- More on scoping later...

```c
int x, y;
void foo(int x)
{
  y = x;
  int y = 0;
  if (y) {
    int y;
    y = 1;
  }
}
```

General View of Binding Time

- Binding time (Scott): the time an answer becomes associated to an open question
- Examples of bindings and binding time
  - Binding of variable to values
    - Dynamic
  - Binding of call to target
    - Static (in C: a call foo())
    - Dynamic (in Java: a.m() )
  - Binding of variable to type
    - Static (in Java, C++, C: declared types for variables)
    - Dynamic (in Smalltalk, Python: actual run-time types of objects)

General View of Dynamic Binding

- Dynamic binding
  - What are the advantages of dynamic binding?
  - Disadvantages?
- An example: Cost of dynamic binding of call to target method in OO languages

```
Example: Cost of Dynamic Dispatch in C++
```

Extra instructions: cost extra!

Other Choices Related to Binding Time

- Pointers: introduce “heap variables”
  - Good for flexibility – allows dynamic structures
  - Bad for efficiency – directly cost: accessed indirectly; indirect cost: compiler unable to perform optimizations!
- Most PLs support pointers
  - Issues of management of heap memory
    - Explicit allocation and deallocation
    - Implicit deallocation (garbage collection)
- PL design choices – many subtle variations
  - No pointers (FORTRAN 77)
  - Explicit pointers (C++ and C)
  - Implicit pointers (Java)

Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- Scoping
  - Static Scoping
  - Dynamic Scoping
Storage Allocation Mechanisms

- Again, note on use of term **object**: anything that can have a name (typically, variable location)
- Static storage – an object is given absolute address which is the same throughout execution
  - What is an example of static data?
- Stack storage – stack objects are allocated on a run-time stack at subroutine call and deallocated at return
  - Needs a stack management algorithm
  - What is an example of stack data?
- Heap storage - long-lived objects are allocated and deallocated at arbitrary times during execution
  - Needs the most complex storage management algorithm

Examples of Static Data

- Static variables
- Program code
- Tables of type data (e.g., inheritance structure)
- Dispatch tables (VFTs) and other tables
- Other

Examples of Stack Data

- What data is stored on the stack?
- Parameters
- Local variables
- Compiler-generated temporaries (i.e., for expression evaluation)
- Bookkeeping (stack management) information

Combined View of Storage

- Stack contains 1 activation record (or stack frame) per executing subroutine.
- Heap contains objects allocated and not yet de-allocated

Run-time Stack

- Stack contains frames of all subroutines which have been entered and not yet exited from
- Frame contains all information necessary to update stack when a subroutine is exited
- Addresses for local variables are encoded as \( fp \) (frame pointer) + offset
- **Idea**: When subroutine is entered, its frame is placed on the stack and \( sp \) (the stack pointer --- i.e., next available location on stack), and \( fp \) (the frame pointer --- i.e., the current frame pointer) are updated. All accesses for local variables use this frame. When subroutine exits, its frame is removed from the stack and \( sp \) and \( fp \) updated.

Frame Details

- Fixed length portion: same length for every subroutine
  - Return address (to code within caller)
  - Miscellaneous bookkeeping information
    - called-by link: pointer to beginning of caller’s frame
    - Saved state (register values of caller)
    - Link for accessing non-local variables
- Variable length portion: length varies by subroutine
  - Local variable storage (including parameters)
  - Compiler-generated temporary storage for subexpressions
Frame Example

\begin{verbatim}
void foo(double rate, double initial) {
  double position; ...
  position = initial + rate*60.0; ...
  return;
}
\end{verbatim}

Assume \texttt{foo()} is called by \texttt{bar()}.  

Frame of \texttt{foo()}

\begin{tabular}{|l|l|}
\hline
\multicolumn{2}{|c|}{Temps} \\
\hline
\hline
\texttt{tmp1} & \texttt{miscinfo} contains: \texttt{position} \\
\hline
\texttt{tmp2} & \texttt{pointer to bar()’s frame} \\
\hline
\hline
\multicolumn{2}{|l|}{Locals} \\
\hline
\hline
\texttt{position} & \texttt{other info} \\
\hline
\texttt{initial} & \texttt{miscinfo} \\
\hline
\texttt{rate} & \texttt{Return address in code of caller} \\
\hline
\multicolumn{2}{|l|}{Return address} \\
\hline
\end{tabular}

Question

- \texttt{sp} (stack pointer) points to the next available location on stack, right after current frame \texttt{Q}
- \texttt{fp} (frame pointer) points to start of current frame \texttt{Q}

- When \texttt{Q}’s routine exits (\texttt{Q} is popped off the stack), what are the new values of \texttt{sp} and \texttt{fp}?  
  - \texttt{sp} (new) = \texttt{fp}
  - \texttt{fp} (new) gets its value from the called-by link of \texttt{Q}

Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- Scoping
  - Static Scoping
  - Dynamic Scoping

Scoping

- 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 - Block Structured PL

\begin{verbatim}
program a, b, c: integer;
    procedure R(a: integer; main.a, main.b, P.c main.P(), P.S(), main.R())
        main.a, main.b, S.c, S.d main.P(), P.S(), S.R(), P.S(), main.P()
    end R;
    R(); R(); R(); end program
\end{verbatim}

Nested block structure allows locally defined variables and subroutines
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).

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

static link (access link; static environment)
Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Example

```c
main()
    a, b, c: integer;/*1*/
    procedure P(){ /*3*/
        c: integer;
        procedure S(){ /*8*/
            c, d: integer;
            procedure R(){ /*10*/
                ...
                } //end R; /*11*/
            R(); /*9*/
        } //end S; /*12*/
    R();/*4*/
    S(); /*7*/
} //end P; /*13*/
procedure R(){ /*5*/
    a: integer;
    = a, b, c;
} //end R; /*6*/
…; P();/*2*/
…
```}

Observations

- The static link always points to the same procedure, no matter where it is called from.
  - Used to implement static scoping using a display.

- The dynamic link points to different procedures, depending on the calling context.

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 resolved to the declaration of that variable in the most recently invoked and not yet terminated frame.
Dynamic Scoping

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

Example

```
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
```

Which a is modified at /*1*/ under dynamic scoping? Z.a or W.a or both?

Example

```
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
```

Is this program legal under static scoping? If so, which a is modified? If not, why not?

Example

```
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?
Exam Review

- Regular Expressions
- CFGs
  - Derivation, parsing, ambiguity, operator precedence and associativity
  - LL(1) grammars and parsing
  - FIRST and FOLLOW sets, LL(1) parsing table
  - Obstacles to LL(1)-ness
  - SLR(1) grammars and parsing
  - CFSM and SLR(1) parsing tables
  - Conflicts in SLR(1)

Spring 16 CSCI 4430, A Milanova

---

**Exam Review**

- **Prolog**
  - Concepts: Search trees, Unification, Backtracking
  - One type of question. Given a Prolog predicate and a query, “Find the first binding” or “Find all bindings”
  - Second type of question. Given a description, write a predicate. 3-4 lines of code.

Spring 16 CSCI 4430, A Milanova

---

**Quiz 1**

**Question 1.** (2pts) Consider the operator grammar

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \times \text{expr} \mid \text{term} \\
\text{term} & \rightarrow \text{id} \mid \text{id} \# \text{id} \mid \text{id} \# \text{id} \# \text{id}
\end{align*}
\]

How many parse trees are there for string \( \text{id} \times \text{id} \times \text{id} \times \text{id} \)?

(a) 0
(b) 1
(c) 2
(d) 5

---

**Question 2.** (2pts) Consider the grammar above. This derivation

\[
\begin{align*}
\text{expr} \rightarrow \text{expr} \times \text{expr} \rightarrow \text{term} \times \text{expr} \rightarrow \text{id} \times \text{expr} \rightarrow \text{id} \times \text{id} \times \text{expr} \\
\quad \rightarrow \text{id} \times \text{id} \times \text{id} \times \text{id}
\end{align*}
\]

is (a) rightmost
(b) leftmost
(c) neither

Spring 16 CSCI 4430, A Milanova

---

**Quiz 2**

**Question 1.** (2pts) Draw a parse tree for string \( \text{id} \times \text{id} \) if \( \text{id} \) then \( \text{id} \) else \( \text{id} \) in the “dangling else” associates with the nearest then.

---

**Question 2.** (2pts) As we discussed in class, this grammar is not LL(1). Specifically, there is conflict in cell \( \text{else-part, else-part} \) as both \( \text{else-part} \rightarrow \text{else-stmt} \) and \( \text{else-part} \rightarrow \) apply on token else. How can you resolve this conflict, so that an else would associate with the nearest unmatched then?

(a) Always choose \( \text{else-part} \rightarrow \text{else-stmt} \).
(b) Always choose \( \text{else-part} \rightarrow \).

---

**Question 3.** (2pts) The grammar is SLR(1).

(a) true
(b) false

---

**Question 4.** (2pts) Given the closure of LH item \( \text{start} \rightarrow \text{else-stmt} \).

Spring 16 CSCI 4430, A Milanova

---

**Quiz 3**

**Question 2.** (2pts) Consider \( \text{gcd}(\text{the Greatest Common Divisor algorithm}) \) in Prolog:

\[
\text{gcd}(A,B,R) \begin{cases} 
\text{gcd}(A,B,R) & : A = B, R = A \\
\text{gcd}(A-B,B,R) & : A > B, A \mod B = A - R, R = B \\
\end{cases}
\]

When \( A = 5, B = 2 \), can we call \( ?- \text{gcd}(A,B,R). \)

(a) yes
(b) no

Spring 16 CSCI 4430, A Milanova
Quiz 3

Question 1. (2pts) Which inference method does Prolog use?

(a) backward chaining
(b) forward chaining

Question 2. (2pts) Consider gcd (the Greatest Common Divisor algorithm) in Prolog:

\[
gcd(A,B,R) :- A = B, R = A. \quad \text{\%base case: when } a=b, \text{ then } gcd(a,b) = a = b.
gcd(A,B,R) :- A > B, A1 is A-B, gcd(A1,B,R). \quad \text{\%when } a>b, \text{ gcd}(a,b) = gcd(a-b,b).
gcd(A,B,R) :- A < B, B1 is B-A, gcd(A,B1,R). \quad \text{\%ditto.}
\]

Is this program “invertible”? (That is, given integers \( b \) and \( d \), can we call \(- gcd(A,b,d) \) to generate a sequence of integers \( a \) s.t., \( gcd(a,b) = d \).)

(a) yes
(b) no

Question 3. (2pts) Recall the classmates Prolog program:

takes(jane, his).
takes(jane, cs).
takes(ajit, art).
takes(ajit, cs).
classmates(X,Y) :- takes(X,Z), takes(Y,Z).

Query ?- classmates(A,B).

has this many answers (each answer is a pair of bindings):

(a) 1
(b) 4
(c) 5
(d) 6

Question 4. (2pts) Now, it’s our favorite grammar again:

1. \( S \rightarrow S \ b \ S \ a \ S \)
2. \( S \rightarrow b \ S \ a \ S \)
3. \( S \rightarrow \ )

Here is one backtracking parser, in Prolog:

\[
p(I,[1|S]) :- append([a|I1],[b|I2],I), p(I1,S1), p(I2,S2), append(S1,S2,S).
p(I,[2|S]) :- append([b|I1],[a|I2],I), p(I1,S1), p(I2,S2), append(S1,S2,S).
p([],[3]).
\]

Query \p{[ b,b,a,b,b,a,b,a,a,b,a,b,a,b,b,a,b,a,b,a ]} . \( \text{\%as in HW2} \) produces this answer first:

(a) \([2, 2, 1, 3, 2, 1, 2, 3, 3, 3, 3, 1, 2, 1, 3, 3, 3, 3, 3]\)
(b) \([2, 2, 3, 2, 2, 3, 2, 3, 2, 3, 3, 3, 2, 3, 2, 3, 2, 3, 3]\)

Quiz 3

Question 4. (2pts) Now, it’s our favorite grammar again:

1. \( S \rightarrow a S b S \)
2. \( S \rightarrow b S a S \)
3. \( S \rightarrow \)

Here is one backtracking parser, in Prolog:

\[
p(I,[1|S]) :- append([a|I1],[b|I2],I), p(I1,S1), p(I2,S2), append(S1,S2,S).
p(I,[2|S]) :- append([b|I1],[a|I2],I), p(I1,S1), p(I2,S2), append(S1,S2,S).
p([],[3]).
\]

Query \p{[ b,b,a,b,b,a,b,a,a,b,a,b,a,b,a ]} . \( \text{\%as in HW2} \) produces this answer first:

(a) \([2, 2, 1, 3, 2, 1, 2, 3, 3, 3, 3, 1, 2, 1, 3, 3, 3, 3, 3]\)
(b) \([2, 2, 3, 2, 2, 3, 2, 3, 2, 3, 3, 3, 2, 3, 2, 3, 2, 3, 3]\)