Programming Languages

Tevfik Koşar

Lecture - XXVIII
May 4th, 2006

Final Exam

- Date: May 11th, Thursday
- Time: 12:30pm - 2:30pm
- Place: 215 TUREAUD HALL

- Final will be comprehensive. It will cover everything we have learned this semester.
- Final will be open book. You can use your book during the exam, but no other notes.
- Check the class notes. Anything not covered in the class will not be asked.
- Expect similar questions to the homework assignments and quizzes.
Hints

• Be sure that you know:
  - how to write regular expressions
  - how to generate NFA and DFA from regular expressions
  - how to generate parse trees
  - difference between top-down and bottom-up parsing; LL vs LR grammars
  - difference between static vs dynamic scoping
  - how to generate attribute grammars; decorate parse trees; and generate syntax trees
  - expression evaluation orders; applicative vs normal-order evaluation
  - basic scheme functions; lists, searching, and scoping in Scheme

Hints (cont.)

- type equivalence (structural and name equivalence)
- type inference
- memory layout for records (structures) and variant records (Unions)
- basic prolog functions, resolution, unification, lists, and arithmetic.
- OOP; object initialization, static vs dynamic binding
- concurrent programming: data parallelism vs function parallelism
- threads vs processes
- shared memory: synchronization
Regular Expressions

- Write a regular expression which captures all odd numbers in the binary format.

Answer: \((0|1)^*1\)

Context Free Grammars

- Write a CFG which captures all odd numbers in the binary format.

Answer:

Odd \rightarrow \text{Digits} \ 1
Digits \rightarrow \text{Digit} \ \text{Digits} \ | \ \epsilon
Digit \rightarrow 0 \ | \ 1
Finite Automaton

- Convert the regular expression \((0^*1^*)^*1\) to a DFA.

1) Convert regular expr to NFA
2) Convert NFA to DFA
3) Optimize

NFA Construction Rules:
- Concatenation
- Alternation
- Kleene Closure

Parse Trees

- Consider the same CFG:
  
  - Odd \rightarrow Digits 1
  - Digits \rightarrow Digit Digits \mid \epsilon
  - Digit \rightarrow 0 \mid 1

- Show the parse tree for input string 1101
- Is it ambiguous?
Attribute Grammars

• Consider the same CFG:
  \[
  \text{Odd} \rightarrow \text{Digits} \ 1 \\
  \text{Digits} \rightarrow \text{Digit} \ \text{Digits} \ | \ \epsilon \\
  \text{Digit} \rightarrow 0 \ | \ 1
  \]

• Show the attribute grammar which would accumulate the decimal value of the binary input string into a \textit{val} attribute of the root of the pare tree.

Answer:

\[
\begin{align*}
\text{Odd} \rightarrow \text{Digits} \ 1 & \quad :\quad \text{Odd.val := Digits.val + 1} \\
\text{Digits} \rightarrow \text{Digit} \ \text{Digits} & \quad :\quad \text{Digits1.val := Digit.val} \times 2^{(\text{Digits.len})} + \text{Digits2.val} \\
 & \quad \quad \text{Digits1.len := Digits2.len} + 1 \\
\text{Digits} \rightarrow \epsilon & \quad :\quad \text{Digits.val := 0} \\
 & \quad \quad \text{Digits.len := 0} \\
\text{Digit} \rightarrow 0 & \quad :\quad \text{Digit.val := 0} \\
\text{Digit} \rightarrow 1 & \quad :\quad \text{Digit.val := 1}
\end{align*}
\]
Scope Rules: Static vs. Dynamic

program scopes (input, output);
var a : integer;

procedure first;
begin
  a := 1;
end;

procedure second;
var a : integer;
begin
  first;
end;

begin
  a := 2; second; write(a);
end.

Scope Rules (cont.)

program scopes (input, output);
var a : integer;

procedure first;
begin
  if (a < 4) a := 4; else a := 1; end;

procedure second;
var a : integer;
begin
  a := 2; first; write(a); end;

procedure third;
var a: integer;
begin
  a := 3; second; end;

begin
  a := 4;
  first; write(a);
  second; write (a);
  third; write(a);
end.
Evaluation Order

- Applicative-order evaluation
  - Evaluate arguments before passing them to subroutine
- Normal-order evaluation
  - Evaluate them only when it is needed

Evaluation Order (cont.)

Consider the following Scheme definition:

```scheme
(define (test x y) (if (= x 3) 1 y))
```

Write the output of the following call to this function

```scheme
(test 3 (/ 5 0))
```

assuming normal-order evaluation is used:

1

assuming applicative-order evaluation is used:

Error: Division by Zero!
Basic Scheme Functions

Show the return values of the following Scheme expressions:

a) \((\text{car} (\text{cdr } `(1 \ 2 \ 3 \ 4)))\)  
   \[2\]

b) \((\text{cdr} (\text{car } `(1 \ 2 \ 3 \ 4)))\)  
   \[()\]

c) \((\text{cons} (\text{car } `(1 \ 2) ) (\text{cdr } `(3 \ 4)))\)  
   \[(1 \ 4)\]

d) \((\text{car} (\text{list } `(1 \ 2) `(3 \ 4)))\)  
   \[(1 \ 2)\]

e) \((\text{cdr} (\text{append } `(1 \ 2) `(3 \ 4)))\)  
   \[(2 \ 3 \ 4)\]

Basic Scheme Functions (cont.)

Write the output of the following Scheme programs:

a) \(\lll ((a \ 2) (b \ 5) (c \ 9))\)
   \(\lll^* (\text{let* } ((a \ 3) (b \ a) (c \ b))\)
   \(\lll^* (\text{let } ((a \ 4) (c \ a) (b \ c)))\)
   \(\lll^* (+ \ a \ b \ c)) \rrl^*\)  
   \[\text{Output: } 10 \ ; \ 4+3+3\]

b) \(\lll ((a \ 2) (b \ 5) (c \ 9))\)
   \(\lll^* (\text{let* } ((a \ 3) (b \ a) (c \ b))\)
   \(\lll^* (\text{let* } ((a \ 4) (c \ a) (b \ c)))\)
   \(\lll^* (+ \ a \ b \ c)) \rrl^*\)  
   \[\text{Output: } 12 \ ; \ 4+4+4\]
Memory Allocation in Arrays & Variants

- Suppose we are compiling for a machine with 1-byte characters, 2-byte shorts, 4-byte integers, and 8-byte reals, and with alignment rules that require the address of every primitive data element to be an even multiple of the element’s size. Suppose further that the compiler is not permitted to reorder fields. How much space will be consumed by the following array?

\[
A : \text{array } [0..9] \text{ of record}
\]
\[
\begin{align*}
& s : \text{short} \\
& c : \text{char} \\
& t : \text{short} \\
& d : \text{char} \\
& r : \text{real} \\
& i : \text{integer}
\end{align*}
\]

OOP (Static vs Dynamic Method Binding)

Consider the following class hierarchy:

```cpp
class person {...}
class student : public person {...}
class professor : public person {...}

void person::print_label() {...}
void professor::print_label() {...}
void student::print_label(int student_id) {...}
```

student s;
professor p;
person *x = &s;
person *y = &p;
person z = p;

Which `print_label` methods will be called upon the following invocations assuming dynamic method binding is in effect:

- `s.print_label();`  →  `person::print_label();`
- `p.print_label();`  →  `professor::print_label();`
- `x->print_label();`  →  `person::print_label();`
- `y->print_label();`  →  `professor::print_label();`
- `z.print_label();`  →  `person::print_label();`
Prolog: Unification

- Pattern-matching process used to associate variables with their values
  - student(joe).
  - :- student(X).
  - X = joe;

- Unification Rules:
  - A constant unifies only with itself
  - Two structures unify iff same functor and same # of arguments, and corresponding arguments unify recursively
  - A variable unifies with anything.

?- a = a.
   yes
?- a = b.
   no
?- foo(a, b) = foo(a, b).
   yes
?- X = a.
   X = a;
   no
?- foo(a, b) = foo(X, b).
   X = a;
   No
?- A = B.
   A = _123
   B = _123
Prolog: Lists

- \([a, b, c]\) can be expressed as (using vertical bar notation):
  - \([a \mid [b, c]]\)
  - \([a, b \mid [c]]\)
  - \([a, b, c \mid []]\)

- `member(X, [X|T]).`
- `member(X, [H|T]) :- member(X, T).`

- `sorted([]).`
- `sorted([X]).`
- `sorted([A, B | T]) :- A <= B, sorted([B | T]).`

Prolog: Lists

\(\text{append}([], A, A).\)
\(\text{append}(H \mid T), A, [H \mid L]) :- \text{append}(T, A, L).\)

?- \text{append}(\{a, b, c\}, \{d, e\}, L).
\(L = \{a, b, c, d, e\}\)
?- \text{append}(X, \{d, e\}, \{a, b, c, d, e\}).
\(X = \{a, b, c\}\)
?- \text{append}(\{a, b, c\}, Y, \{a, b, c, d, e\}).
\(Y = \{d, e\}\)

\(\Rightarrow\) Prolog predicates do not have a clear distinction between input and output arguments!
Prolog: Arithmetic

- Built-in `is` predicate: unifies its first argument with the arithmetic value of its second argument.

```prolog
?- is(X, 1+2).
X = 3
?- X is 1+2.
X = 3
?- 3 is 1+2.
yes
?- 1+2 is 3.
no
?- X is Y.
<error>
?- Y is 1+2, X is Y.
X = 3
Y = 3
```

Concurrency: Data Parallelism

Decompose DATA into pieces. All the tasks perform the same computation operating with their own piece of data.

```
for (i = 1; i<=60; i++)
{
    a(i) = b(i) * c(i);
}
```

```
for (i = 1; i<=20; i++)
{
    a(i) = b(i) * c(i);
}
```

```
for (i = 21; i<=40; i++)
{
    a(i) = b(i) * c(i);
}
```

```
for (i = 41; i<=60; i++)
{
    a(i) = b(i) * c(i);
}
```
Concurrency: Function Parallelism

- $a = b + 1$
- $a = a + c$
- $a = 2 \cdot a$
- $d = 5 \cdot d$
- $e = d - 1$
- $e = e + 1$
- $g = f^3$
- $f = f - 1$
- $f = f \cdot 4$

Decompose COMPUTATIONS into pieces (functions).
Data are taken to the tasks where they are needed.