Final Exam Study guide

Final Exam Study guide

CS 440: Programming Languages and Translators

Final Exam is 8:00 - 10:00 am, Wed, May 8, 2019

You can bring one sheet (both sides) of notes to the exam. Be sure to include homeworks as you study for the exam.

Exam 1 Material (roughly 25% of exam)

• Haskell: Especially review
  • Given a function definition, what is the function's type?
  • Given function's type, is ... a type-correct expression?
  • What are higher-order functions? How do you recognize them? Examples? Non-examples?

• Regular Expressions
  • Given an English description of a pattern, design a regular expression.
  • Given a regular expression, describe in English what it matches.
  • Does a given regular expression match a given string?
  • Do two regular expressions match the same strings?

Exam 2 Material (roughly 25% of exam)

• NFA, FSM, Scanners
  • Convert NFA → equivalent DFA

• Grammars, Parse trees, Productions
  • Why do we use context-free grammars (not other kinds of grammars) to describe languages?
  • What does it mean to have an ambiguous grammar?
  • What’s the difference between saying a grammar is ambiguous versus saying a language is ambiguous?

• Recursive Descent Parsing
  • Given a grammar, what are the First and Follow sets for its nonterminals?
  • What properties do the First and Follow sets need to have for a language to be LL(1)?
  • Probably won't ask about building an LL prediction table or using a prediction table to trace the parse of an input.

1 Say ± 5% each for Exams 1 or 2 and ± 10% for material after Exam 2.
Final Exam Material (roughly 50% of exam)

1. Bottom-up Parsing
   a. Which gets to use more information about the language as it parses: A top-down parser or a bottom-up parser? Why?
   b. A bottom-up parse corresponds to finding what kind of derivation?

2. LR Parsing
   2.1 LR(0) items
      a. What is an LR(0) item? What does one mean?
      b. Given a grammar, give a list of all its LR(0) items.
   2.2 Nondeterministic Characteristic Finite State Machine (CFSM) for LR parser
      a. When do we have a transition from one LR(0) item to another?
      b. When do we have a shift-shift conflict?
      c. When do we have an LR(0) item with no transitions out of it?
   2.3 Deterministic CFSM
      a. Why do we convert a nondeterministic CFSM to a deterministic one?
      b. What does it mean to take the closure of an LR(0) item? Why do we do this?
      c. How do you recognize the following kinds of conflicts in the deterministic CFSM?: shift-shift, shift-reduce, reduce-reduce
   2.4 LR(0) parser
      a. For an LR(0) parser, when and how do you decide to do a reduction?
   2.5 SLR(1) parser
      a. For an SLR(1) parser, when and how do you decide to do a reduction?
      b. Relative to CR(0) parsers, what advantage(s) / disadvantage(s) does an SLR(1) parser have?
      c. For an SLR(1) parser, what is the Action Table? The Go To table? How do you develop them given the deterministic CFSM?
   2.6 CLR(1) parser
      a. How do the states of a CLR(1) parser differ from the ones for an LR(0) or SLR(1) parser?
      b. For a CLR(1) parser, when and how do you decide to do a reduction?
      c. Relative to SLR(1) parsers, what advantage(s) / disadvantage(s) does a CLR(1) parser have?
   2.7 LALR(1) parser
      a. When and how do LALR(1) parser states differ from CLR(1) parser states?
      b. Relative to CLR(1) parsers, what advantage(s) / disadvantage(s) does a LALR(1) parser have?

3. Substitution, Unification
   3.1 Substitutions, general/special
      a. What do you do substitutions on? What do substitutions do?
      b. What does it mean for one substitution to be more general or more special than another?
      c. Given a substitution to a term, what is the result of applying the substitution to the term?
      d. Given two terms, what substitution was done to get from the first to the second?
      e. What is the code for carrying out a substitution?
3.2 Unification: equation, problem
   a. What does it mean to unify two terms?
   b. What is a unification equation? What is a unification problem?
   c. What does it mean to have a solution to a unification problem?
   d. How do most-general unifiers fit in here?
   e. Be able to say what the unification algorithm does with X = X, X = Y, and X = term.
   f. Given a unification problem, does the problem have a solution? If so, what is it?

4. Monadic Parsing
4.1 Non-monadic code. Assume we have types State and ParseT defined. For the parser
   \[
   \text{p s0 = p0 s0 `yield` } \langle ts1, s1 \rangle \rightarrow \\
   \text{ p1 s1 `yield` } \langle ts2, s2 \rangle \rightarrow \\
   \langle ts1 ++ ts2, s2 \rangle
   \]
   a. What are the types of p0, s0, and each ts1?
   b. Can p s0 return Nothing? If so, when does this happen?
   c. What is the likely definition of yield?

4.2 Monadic values and notation. Assume we're using monad Maybe. For the parser
   \[
   \text{p = p0 `bind2` } \langle ts1 \rightarrow \\
   \text{ p1 `bind2` } \langle ts2 \rightarrow \\
   \text{ mreturn (ts1 ++ ts2)}
   \]
   a. What are the types of p1 and ts1?
   b. What are the types of bind2 and mreturn?
   c. Can we call mreturn on a single parse tree? On []?
   d. Can p, given some input, return Nothing? If so, when does this happen?

4.3 Monadic value using case. Consider the parser
   \[
   \text{p s0 = case p0 s0 of} \\
   \text{ Nothing -> Nothing} \\
   \text{ Just(ts1, s1) -> case p1 s1 of} \\
   \text{ Nothing -> Nothing} \\
   \text{ Just(ts2, s2) -> Just(ts1 ++ ts2, s2)}
   \]
   a. What is the difference in processing done by the parser above and the one in the previous question?
   b. In particular compare the types of the p's, s's, ts's and the value of p s0.

5. Semantic Analysis
5.1 Basis for Semantic Analysis
   a. Where does semantic analysis fit into a compiler?
   b. What operations might we typically do during semantic analysis?
5.2 Attributes
a. What is an attribute and how do we refer to them?
b. What is an attribute rule? How might we write one for the rule \( X \rightarrow A X \)?
c. What is an attribute grammar?
d. What is a synthesized attribute? Give an example.
e. What is an inherited attribute? Give an example.
f. Is the parser state a synthesized or inherited attribute?

5.3 Examples of Attributes; for \( List \rightarrow^* (u, v, (x, y, a), a) \), what kinds of attributes are given below? (Synthetic? Inherited? Neither?)
a. \( List \) has 4 parentheses
b. The \( y \) is surrounded by two pairs of parentheses
c. \( List \) has two \( a \)'s
d. There are two \( a \)'s to the right of \( y \)
e. The second \( a \) has three members to its left

5.4 Attributes in monadic parser. For the parser
\[
p x0 = p0 x0 >>= \langle x1, y1 \rangle ->
p1 x1 >>= \langle x2, y2 \rangle ->
mreturn (x2, combine(y1, y2))
\]
a. Which of the \( x \)'s and \( y \)'s are inherited properties?
b. Which are synthetic properties?

6. IO Monad
a. Why does Haskell use \( IO \) for doing I/O?
b. Why isn't there any useful way to go from an \( IO \) type to a non-\( IO \) type?
c. do block syntax: \( do \ x \leftarrow e1 \ in \ e2 \) is equivalent to ??
d. Skip: Haskell Monad, Functor, Applicative typeclasses

7. Prolog
a. In what sense is Prolog based on logic?
b. How do the following work in Prolog: datatypes, terms, facts, rules, database, queries?
c. How does program execution in Prolog differ from that in, say C? execution as proof discovery
d. What does it mean when we say that Prolog can often execute a program in two directions?
e. Say we're trying to prove \( q \) and we have rules \( q :\sim p \) and \( p :\sim r1, r2 \). How does backward chaining work?
f. Name two imperative features of Prolog
g. Be able to say what the result of a query would be for a simple example such as ancestor / parent; likes and dislikes; graph search; Range test; 2-way execution
h. What is the difference between a rule \( q :\sim p \) and \( q :\sim p, \, ! \)?
i. What is a green cut? A red cut?
Answers to Selected Questions

1. Bottom-up Parsing
   a. A bottom-up parser, because it can use the information of the already-parsed nonterminals to decide which rule to use.
   b. Rightmost derivations in reverse

2. LR Parsing
   2.1 LR(0) item
      a. An indicator of where we are in a parse: \( A \rightarrow \alpha \cdot \beta \), \( A \rightarrow \alpha \cdot \beta \), and \( A \rightarrow \alpha \beta \cdot \) indicate that we're just beginning / in the middle of / have finished using the rule \( A \rightarrow \alpha \beta \) to parse an \( A \).

2.2 CFSM
   a. \( A \rightarrow \alpha \cdot X \beta \) transitions to \( A \rightarrow \alpha X \cdot \beta \) on input \( X \). (Note \( X \) can be a terminal or nonterminal.)
   b. When we have \( A \rightarrow \alpha \cdot X \beta_1 \) and \( A \rightarrow \alpha \cdot X \beta_2 \)
   c. When we have \( A \rightarrow \alpha \cdot \)

2.3 Deterministic CFM
   a. Deterministic FSMs are easier to execute than nondeterministic ones.
   b. The closure of \( A \rightarrow \alpha \cdot B \beta \) includes all the rules \( B \rightarrow \cdot anything \) plus any rules in the closure of \( those \) rules. ...
   c. There aren't shift-shift conflicts. Shift-reduce: you have \( A \rightarrow \alpha \cdot X \beta \) and \( B \rightarrow \gamma \cdot \) in the same state. Reduce-reduce: You have \( A \rightarrow \alpha \cdot \) and \( B \rightarrow \beta \cdot \) in the same state.

2.4 LR(0) parser
   a. LR(0), you always reduce if you can.

2.5 SLR(1) parser
   a. In SLR(1), you reduce \( A \rightarrow \alpha \cdot \) if the lookahead character is in \( \text{Follow}(A) \). You reduce \( A \rightarrow \epsilon \cdot \) if the lookahead character is in \( \text{First}(\beta) \) for some rule \( B \rightarrow \alpha \cdot A \beta \)
   b. SLR(1) parsers can make more refined decisions as to when to reduce than LR(0) parsers.
   c. For state \( s \) and terminal \( b \), Action table \((s, b) = \text{shift } t \) (shift \( b \) and go to state \( t \)) or \text{reduce } n \) (reduce using rule number \( n \)). For nonterminal \( A \), \( \text{GoTo}(s, A) = t \) means that you should go to state \( t \). The deterministic CFM is built from the nondeterministic one using the same algorithm for the NRA \( \rightarrow \) DFA translation.

2.6 CLR(1) parser
   a. A CLR(1) parser has states that look like \((A \rightarrow \alpha \cdot \beta, b)\), which means that if you complete parsing and get to \((A \rightarrow \alpha \beta \cdot, b)\), you should reduce only if the lookahead character is \( b \). LR(0) and SLR(1) states don't include this lookahead.

2.7 LALR(1) parser
   a. Under certain conditions, LALR(1) parsers can merge states \((A \rightarrow \alpha \cdot \beta, b)\) and \((A \rightarrow \alpha \cdot \beta, c)\) and include them in the same deterministic FSM state.

3.1 Substitutions, general/special
Term $t_1$ is more general than $t_2$ if there's a substitution that takes $t_2$ to $t_1$. (For some $\tau'$, $t_1 = t_2 \tau'$.) Substitution $\sigma$ is more general than $\tau$ if for every term $t$, the term $t \sigma$ is more general than $t \tau$. (I.e., there's some substitution that takes $t \tau$ to $t \sigma$.)

3.2 Unification: equation, problem

a. Unifying terms $s$ and $t$ means finding a substitution $\sigma$ where $s \sigma = t \sigma$. I.e., substitution using $\sigma$ takes both $s$ and $t$ to exactly the same term.

b. A unification equation has the form $\text{term}_1 = \text{term}_2$. A unification problem is a set of unification problems.

4. Monadic Parsing

4.1 Non-monadic code

a. The $p$'s are of type $\text{State} \rightarrow ([\text{ParseT}], \text{State})$. The $s$'s are of type $\text{State}$ (i.e., $\text{String}$). The $t$'s are lists of parse trees, so $[\text{ParseT}]$.

b. $p \circ 0$ cannot return $\text{Nothing}$

c. $\text{yield} x f = f x$

4.2 Monadic values and notation

a. Let type $\text{Parser}$ be $(\text{State} \rightarrow \text{Maybe}([\text{ParseT}], \text{State}))$, then the $p$'s :: $\text{Parser}$. The $t$'s :: $[\text{ParseT}]$.

b. $\text{bind}_2 :: \text{Parser} \rightarrow ([\text{ParseT}] \rightarrow \text{Parser}) \rightarrow \text{Parser}$. $\text{mreturn} :: [\text{ParseT}] \rightarrow \text{Parser}$

c. $\text{mreturn}$ $\text{tree}$ is not type-correct. ($\text{mreturn} [\text{tree}]$ would be ok.) $\text{mreturn} [\text{ }]$ is ok.

d. Given an input, $p$ will generate $\text{Nothing}$ if either parsers $p0$ or $p1$ return $\text{Nothing}$ on their inputs.

4.3 Monadic value using case

a. The code takes the previous problem's code and substitutes the definitions of $\text{bind}_2$ and $\text{mreturn}$. The two parsers should behave identically.

b. All the types are the same across the two parsers; they also return the same results.

5. Semantic Analysis

5.1 Basis for Semantic Analysis

a. Semantic analysis happens just after parsing

b. Semantic analysis typically includes operations like typechecking and checking for declarations of variables, which are properties we can't write using context-free grammars.

5.2 Attributes

a. An attribute is a property. A reference like $A\.\text{value}$ indicates that value is a property of nonterminal $A$.

b. An attribute rule says how to calculate an attribute. For $X \rightarrow A X$, we add subscripts to distinguish between uses of the same nonterminal: $X_0 \rightarrow A X_1$. A rule might be $X_0\.\text{property} 1 = \text{combination of } A\.\text{property} 2$ and $X_1\.\text{property} 3$.

c. Parser state is an inherited property

5.3 Examples of Attributes

a. Synthetic property of $\text{List}$
b. Inherited property of nonterminal generating \( y \)
c. Synthetic property of \( \text{List} \)
d. Neither synthetic nor inherited property of nonterminal generating \( y \)
e. Inherited property of nonterminal generating second \( a \)

5.4 Attributes in monadic parser
a. All the \( x \)'s are inherited properties.
b. All the \( y \)'s are synthetic properties.

6. IO Monad
a. Haskell can’t have I/O operations produce regular values like integers because I/O has side effects, which breaks referential transparency. Having no way to go from \( \text{IO} \ t \) to a non-\( \text{IO} \) type keeps us from being able to see that referential transparency has been broken.
b. \( \text{do } x \leftarrow e_1 \text{ in } e_2 \) is equivalent to \((e_1 >>= \lambda x \rightarrow e_2)\)

7. Prolog
c. Prolog program execution involves proof-searching, not explicit state manipulation (as in assignment states, e.g.)
d. Two-way execution: In Prolog, with a term like \( p(x, y) \), it might be possible to calculate \( y \) given \( x \) but also possible to calculate \( x \) given \( y \).
e. If we want to prove \( q \), it’s sufficient to prove \( p \) because of the rule \( q : - p \). From the rule \( p : - r_1, r_2 \), we find that to prove \( q \), it’s sufficient to prove \( r_1 \) and \( r_2 \).
f. Fact/rule order in the database (top to bottom) and term order in rules (left to right). Cuts are a third imperative feature.
h. With the \( ! \) (cut), \( q : - p, ! \) says that if we prove \( q \) using this rule, then we should never try backtracking to re-prove \( p \) in order to find another proof of \( q \).
i. A green cut doesn’t affect the set of outputs for a program; it simply keeps us from useless reproving of things. A red cut does change the set of possible answers from a program.
Review: Monadic Parsing, Semantic Analysis

Combinatorial and Monadic Parsing
All of the parsers below parse the same language and build the same parse trees; they differ by what the parser code show explicitly vs implicitly, and they differ in how the parser code is presented.

First, let's look at non-monadic parsers; parser type \(P\) is \((s \rightarrow ([t], s))\)

All the example code shows running two parsers one after another (basically, parser &> parser).
Note: The code is actually just sketches: I've left off subscripts in \(var_0, var_1\), etc.,

**Notation:** \(t\) - parse tree; \(ts\) - \([t]\); \(s\) - state; \(p\) - parser, (so \(p :: P\))

Non-Monadic Results
1. Normal function call; use let to access \(t, s\)
   
   ```
   let (ts, s) = p s
   (ts, s) = p s
   in (bt ts ts, s) -- bt builds a tree from subtrees
   ```

2. Reversed function call; use \(\lambda\)
   
   ```
   p s `yield` \((ts, s)\) -> -- x `yield` f = f x
   p s `yield` \((ts, s)\) ->
   (bt ts ts, s)
   ```

Monadic results
The parser type \(P\) is \((s \rightarrow M([t], s))\) [writing \(M\) for \(Maybe\); generically \(M\) would just be a monad]

3. Normal function call, use ```case``` to analyze structure [here \(M\) definitely is \(Maybe\)]
   
   ```
   case p s of
   Nothing -> Nothing
   Just(ts, s) -> case p s of
   Nothing -> Nothing
   Just(ts, s) -> Just(bt ts ts, s)
   ```

4. **Combinatorial parser** [compare with Lecture 11]. Here we Hide \(s\); Hide \((t, s)\)
   
   ```
   -- (&>) :: P -> P -> P; btt :: P -> P
   ```
   
   ```
   bt (p &> p)
   -- bt needs to know how to combine the result of running
   -- (p &> p) on input state to get final \(M([t], s)\) result
   ```
Monadic Parsers
Parser type \( P \) is \((s \rightarrow M(t, s))\)

Simple Monadic Operation (value >>= ...)
5. Show \( t, s \) parameters and results [compare with Lecture 19]. The bind here is monadic, but it works on concrete \( M(t, s) \) values. The parsers after this one work on higher order functional types that yield \( M(t, s) \) values (or some version of \( M \)).

\[
\begin{align*}
\text{bind} & : M(t, s) \rightarrow ((t, s) \rightarrow M(t, s)) \rightarrow M(t, s) \\
\text{bind} \text{ Nothing} & = \text{Nothing} \\
\text{bind} \text{ (Just} (t, s) \text{)} & f = f(t, s) \\
\text{return} & = \text{Just} \\
\end{align*}
\]

\[
\begin{align*}
p \ s & \text{`bind`} \ (t, s) \rightarrow \\
p \ s & \text{`bind`} \ (t, s) \rightarrow \\
\text{return(bt} \ t \ t, s) & \quad -- \text{this bt} :: t \rightarrow t \rightarrow t \\
\end{align*}
\]

Monadic Operations for (function >>= ...)
\[
(\gg=) : : P \rightarrow (t \rightarrow P) \rightarrow P \quad \text{-- was bind2 in Lecture 20}
\]
\[
p \gg= f = s \rightarrow \text{case } p \ s \text{ of} \\
\quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \text{Just} (t, s) \rightarrow f \ t \ s
\]

\[
\begin{align*}
\text{return} & :: t \rightarrow P \quad \text{[was mreturn in Lecture 20]} \\
\text{return} \ t \ s & = \text{Just}(t, s)
\end{align*}
\]

6. Monadic parser that hides \( s \), shows \( t \) [compare with Lecture 20]

\[
\begin{align*}
p \gg= & \ t \rightarrow \\
p \gg= & \ t \rightarrow \\
\text{return(bt} \ t \ t) & \quad -- \text{this bt} :: t \rightarrow t \rightarrow t
\end{align*}
\]

Monadic Parsers with Attributes

Monadic Parser with Synthetic Attribute
Replace \( t \) with \((t, syn)\). Parser type \( P \) is \((s \rightarrow M((t, syn), s))\). [Compare with Lecture 22]

7. Monadic parser that hides \( s \) and shows \((t, syn)\)

\[
\begin{align*}
\gg= & : : P \rightarrow ((t, syn) \rightarrow P) \rightarrow P \\
\text{return} & :: (t, syn) \rightarrow P \\
\end{align*}
\]
\[
\begin{align*}
p \gg= & \ (t, syn) \rightarrow \\
p \gg= & \ (t, syn) \rightarrow \\
\text{return(btt} \ (t, syn) (t, syn)) & \quad -- \text{btt} :: (t, syn) \rightarrow (t, syn) \rightarrow (t, syn) \\
\text{-- btt builds a a tree and syn result using tree & syn of subtrees}
\end{align*}
\]
Combinatorial Parser with Inherited Attribute

8. Monadic parser that hides $s$, $t$, and $syn$. This would be a combinatorial parser like (4) except that now $btt$ needs to know how to combine two ($t, syn$), not two $t$.

\[
\text{bt} \ (p \ &> p)
\]

\[
\text{-- } (>&) \ :: \ P \ -> \ P \ -> \ P; \ btt \ :: \ P \ -> \ P
\]

Monadic Parser with Inherited Attribute

Like the state, an $inh$ attribute gets passed to / returned by a parser.

9. Parser that shows $t$ and $inh$ and hides $s$ [compare with Lecture 22]

The parser type $P$ is ($inh \rightarrow Q$) where $Q$ is ($s \rightarrow M((t, inh), s)$).

For return, $bt$ and $bh$ have to build new tree / new $inh$ respectively from ones for subtrees

\[
\text{p inh >>= } \ \backslash(t, inh) ->
\]

\[
\text{p inh >>= } \ \backslash(t, inh) ->
\text{ return (bt t t, bh inh inh)}
\]

Monadic Parser with Inherited Attribute as Part of State

The parsers above run with the state type $s = String$, but you can include an inherited attribute and use ($s, inh$), and the basic >>= and return routines work okay. Various base parsers will need to change to use/modify the $s$ or the $inh$ part of state. (E.g., increasing an inherited size attribute would be added to parsing an identifier.)

10. Parser that shows $t$ and hides $s$ and $inh$. This is the same code as for #6 above, but parser type $P$ is now ($s, inh$) -> $M(t, (s, inh))$

\[
\text{p >>=} \ \backslash t ->
\]

\[
\text{p >>=} \ \backslash t ->
\text{ return(bt t t) } \quad \text{-- this bt :: t -> t -> t}
\]