
CS 320 Principles of Programming Languages, Spring 2019 Homework 3

Notes and sample solutions

Question 1:

We provide multiple examples in each of the three program categories below, but students were only expected to provide two examples in each case. It is not necessary for students to offer the same examples described here, so long as the descriptions match the different program categories and describe tools that are specific to applications involving .prop files.

PROGRAM ANALYSIS: We are looking for tools that take a .prop file as input and produce some useful results. Examples:

a) A visualization tool that will generate a diagram showing the structure of the input circuit. This might be used to inspect and understand the structure of a complex circuit for (and perhaps to reveal flaws in the way that its components of a circuit are connected).

b) A circuit simulator (i.e., an interpreter) that calculates the output values that will be produced when a given circuit is supplied with specific inputs. This might be used to check the behavior of a circuit on specific inputs, or to build a truth table to show its behavior on all possible inputs.

c) A bug finding tool that will read .prop files and alert the user to basic syntax errors, or to circuit descriptions that do not provide definitions for all of the variables named as outputs, etc. This might be used to identify (and then remove) errors in a circuit file.

d) An import checking tool that reports an error if the import section of a .prop file refers to non-existing files, or to circuits that are not mentioned in the rest of the file, or if the definitions in the rest of the file use circuits that are not listed in the imports. This tool would have similar uses to (c).

e) A gate count tool that determines the total number of primitive AND, OR, and NOT gates that would be required to build a complete circuit. This might be used, for example, to check that a

proposed circuit design will satisfy engineering requirements for size, power consumption, cost, etc.

f) A gate delay tool that determines the maximum gate delay for a circuit (i.e., the maximum number of gates that any input signal will need to pass through to determine the value of one of the output signals.) This might be used to check, for example, that a proposed circuit will meet design constraints for worst-case timing behavior.

g) A satisfiability tester that will try to determine if there are any combinations of inputs that will produce a true output for each wire. This tool might be used to identify redundant circuits; if there is no way that a given output will ever be true, then the circuit for that output could be replaced with a direct connection to FALSE.

h) A circuit comparison tool that will take two circuits as inputs and compare them, perhaps using testing or equivalences, or some combination of techniques, to see if the two designs are equivalent. This might be used to determine whether the optimized circuit implementation developed by a skilled engineer behaves in the same was as a less efficient "reference implementation".

PROGRAM SYNTHESIS: We are looking for tools that take some data/parameters as input and generate a .prop file as output.

a) A generator for common circuits. For example, we might have a program that takes a number n as input and that generates the definition of an ANDn circuit for an AND gate with n inputs. A tool like this would save users from having to do the tedious and error prone work of constructing these circuits manually.

b) A function generator that takes a truth table as input and generates a corresponding circuit as output. Using a tool like this would save a circuit designer from having to figure out all the details of what gates are needed to implement a logic function that has been specified by a truth table, increasing the designer's productivity and, assuming the tool is implemented correctly, avoiding mistakes that might be made if the task were done by hand.

c) A template generator might be used to generate circuit files of a particular form with gaps, marked for example by comments or other markers, that can then be filled in by hand to complete

the design. This kind of tool might help circuit designers to construct .prop files more quickly in situations where there is a lot of boilerplate that can be generated automatically.

PROGRAM TRANSLATION: We are looking for tools that take programs as input and generate programs as outputs. These "programs" might be in the form of .prop files that describe circuits, but they could also be in other languages.

a) A compiler that translates a circuit description in to C code that can then be compiled and executed at high speed to test the behavior of the circuit on a large collection of possible inputs.

b) An optimizer that rewrites a circuit to simplify its structure. A tool like this might: use equivalences to reduce the number of gates that are needed; detect places where the same result is computed by two different circuits and then sharing the result of one circuit instead; and identify portions of circuits that can be removed because their outputs are not used. This would be useful in preparing a circuit for a practical application (e.g., reducing the cost of building or the power consumption of a physical version of the circuit).

c) A partial evaluator that takes a circuit as input as well as specific values for some of its inputs, and then generates a new specialized version of the input. For example, given a general circuit with two n-bit numbers as inputs (i.e., 2*n input parameters) and returns True if the numbers are equal, we could generate a simplified circuit in which one of the numbers is a known constant. (For example, turning an are-these-numbers-equal function into an is-this-number-zero function.)

d) A compiler that translates a circuit description in a .prop file into a hardware description language (like Verilog or VHDL). This would be useful as a step towards manufacturing a chip or producing an FPGA implementation of a circuit.

e) An update tool that generates a new version of a circuit file by replacing references to certain import files with the names of new, more efficient implementations for those circuits. This might be useful when trying to update a large set of .prop programs to use a new library of basic circuit building blocks.

Question 2:

---

a) ERRORS DETECTED DURING SOURCE INPUT: These are errors that would be detected in the process of trying to read the underlying raw representation of a program. It is often quite difficult to write down errors of this kind---if the compiler cannot read the input as a sequence of characters, then it is likely to be difficult for us to write it down in that way here. For that reason, we will allow answers that do not include an example so long as there is some brief justification for omitting the example.

- An error would occur if there is no file on the user's computer corresponding to the name of a circuit that is specified as input to the front end. Obviously, it is not possible to show a sample program for this kind of error.

- An error would occur if the specified input file cannot be read because of corruption on the disk where it is stored. Clearly there is nothing the front end can do about this, but it must still be prepared to report an error if a problem like this is detected. Again, it is not possible to show a concrete program for an error of this kind.

- If the input file contains a character that is not legal in any .prop file, then this could be detected and reported as an error during source input. Note that a character like '/' cannot be treated as an illegal character: even though it cannot be used in any valid token, it is permitted (indeed, required) in any comment.

  Example.prop:

```
    ^Z      // assuming that control Z is an illegal character!
```

---

b) ERRORS DETECTED DURING LEXICAL ANALYSIS: We are looking for situations where the input can be read as a valid sequence of characters, but it is not possible to group those characters as a valid sequence of tokens. In these cases, we can demonstrate each error with a very short input file by triggering a lexical few characters in a file.

- An unterminated comment; the individual characters are valid, but we cannot construct a valid sequence of tokens for this program:

Example.prop:

```
/* this comment does not have an end ...
```

- Numeric literals are not valid tokens in the language described in the preamble (the individual digit characters are valid):

Example.prop:

```
1    // oops, 1 is not a valid token!
```

- The symbol / is not a valid token (although // can be used to start a comment, so we know that / should not be treated as an illegal character in source input!)

Example.prop:

```
/   // if we remove the spaces, this would be valid
```

---

c) ERRORS DETECTED DURING PARSING: We are looking for examples where the input can be interpreted as a valid sequence of tokens but the tokens do not match the grammatical rules of the language. There are numerous ways to produce errors of this kind; what follows is NOT an exhaustive list! Note that all of the programs shown in the examples below are comprised of a valid sequence of tokens, so all of them will be accepted by lexical analysis.

- An empty file (or a file containing only comments and spaces) corresponds to an empty list of tokens, so it passes lexical analysis, but not parsing because it does not begin with a "circuit" token.

```
// empty file
```

- A file containing only valid tokens will not match the grammar if the first word is not "circuit":

```
sirkit A   // "sirkit" and "A" are valid variable names
```

- A file containing only valid tokens could be missing a name for the circuit. (There are many similar examples such as missing names in import definitions, missing semicolons, etc.)

```
circuit ;    // there should be a name before the ;
```

- The individual Prop expressions in a circuit file must all be valid. For example, every use of AND should have two inputs.

```
    circuit C A;  // all tokens here are valid ...
    Out = AND A;  // ... but oops, we are missing argument for A
```

- Many more examples could be added here (e.g., mismatched parentheses, keywords used where variable names are required, etc.)

---

d) ERRORS DETECTED DURING STATIC ANALYSIS: We are looking for programs that can be derived from the language grammar but that do not correspond to valid circuits. Again, there are quite a few ways to produce errors of this kind, so what follows is NOT an exhaustive list! Note that all of the programs shown in the examples below are valid syntax according to the grammar given in the program, so all of them will be accepted by the parser.

- The same input variable name cannot be used multiple times:

```
    circuit C A A;    // cannot use A twice
```

- The circuit description must provide a definition for Out:

```
    circuit C A;      // no definition for Out
```

- The body of the circuit must not refer to a variable that is not defined or specified as an input.

```
    circuit C A;
    Out = B;          // no variable called B here!
```

- The body of a program must not contain multiple definitions for a single variable (or a definition for one of the inputs):

```
    circuit C A;
    Out = A;
    Out = NOT A;     // two definitions for Out
```

- The value of a variable cannot be specified by a recursive definition, corresponding to a circuit with a loop (i.e., to a circuit that we cannot represent with a finite abstract syntax tree).

```
    circuit C A;
    Out = AND Out A; // Out is used as an input and output
```

- etc...

Question 3:

In general, if a program generates an error diagnostic, then it must be triggered by one of three things:
- an error in the input to the program;
- an error in the program itself; or
- an error that occurs when the program generates output.

For the scenario described in this question, we are working with a back end that receives its input from the front end of a compiler, whose output is a *validated* structured representation of the source program. This means that the input to the back end must be a valid program. Assuming also that the back end itself is implemented correctly (ensuring that it translates any valid input to a valid output), we would not, in principle, expect any errors to occur during this part of the compiler.

The only remaining possibility is that an error could occur if, for reasons beyond the control of the compiler, it is unable to generate the expected output. For example, although it is unlikely to happen very often in practice, the developers of the back end will likely need to include some small amount of code to deal with the possibility that there will not be enough memory to store the generated code, or that an attempt to write an output file to disk will fail because (for example) the disk is full.

Question 4:

We present a description of the changes that are needed to implement a parser for lists of definitions, following the three part structure specified in the question.

CHANGES IN LEXICAL ANALYSIS: We need to add two new tokens, one for semicolons and one for equal signs, and we can use exactly the same procedure for this as in the lab exercises. First, we need to add new constructors to the Token type:

```
> data Token = ...
>            | TEQ   | TSEMI
>              ...
```

And then we need to add new equations in the definition of lexProp

to return those token values when the corresponding characters are found in the input stream:

```
> ...
> lexProp ('=':cs)  = TEQ    : lexProp cs
> lexProp (';':cs)  = TSEMI  : lexProp cs
> ...
```

CODE FOR THE DEFINITION OF parseDefns: Before we can build a parser, we need to have a clear understanding of the language that it is recognizing. Fortunately, the question already provides a specific grammar for lists of definitions:

```
  Defs    -> Def ";" Defs
  Defs    ->
  Def     -> name "=" Prop
```

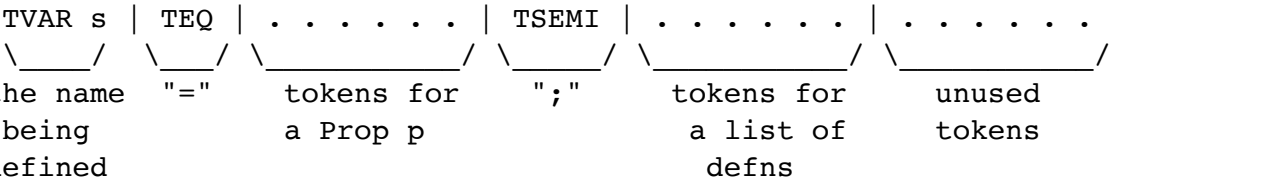
In fact we can simplify this a little further (which is usually a good thing to do because a simpler grammar will often result in a simpler parser):

```
  Defs    -> name "=" Prop ";" Defs
  Defs    ->
```

Our goal now is to define a parser of the following type:

```
> parseDefns :: [Token] -> ( [(String, Prop)], [Token] )
```

The first of the productions for Defs above requires us to break an input sequence of tokens into the following sections:



Using the patterns covered in labs, we can build a suitable equation for parseDefns as follows:

- Step 1: match the initial TVAR:

```
  parseDefns (TVAR s : ...) = ...
```

- Step 2: match the following EQ symbol:

```
  parseDefns (TVAR s : TEQ : ...) = ...
```

- Step 3: use parseProp to extract a Prop from the tokens after TEQ:

```
  parseDefns (TVAR s : TEQ : ts)
    = case parseProp ts of
        (p, ts1) -> ...
```

- Step 4: check for semicolon after the definition:

```
  parseDefns (TVAR s : TEQ : ts)
    = case parseProp ts of
        (p, TSEMI : ts1) -> ...
        (p, ts1) -> error "missing semicolon"
```

- Step 5: parse a list of definitions ds from the remaining tokens (using a recursive call to parseDefns):

```
  parseDefns (TVAR s : TEQ : ts)
    = case parseProp ts of
        (p, TSEMI : ts1) -> case parseDefns ts1 of
                              (ds, ts2) -> ...
        (p, ts1) -> error "missing semicolon"
```

- Step 6: combine the first definition (s, p) with the rest of the definitions in ds to make a list (s, p) : ds as the final result:

```
> parseDefns (TVAR s : TEQ : ts)
>   = case parseProp ts of
>       (p, TSEMI : ts1) -> case parseDefns ts1 of
>                             (ds, ts2) -> ((s,p):ds, ts2)
>       (p, ts1) -> error "missing semicolon"
```

This handles the case for input token streams that begin with a definition, but we also need to provide code that will work for token streams that do not start with a definition. This corresponds to the second production in our grammar. In this case, we can just return an empty list of definitions as the result:

```
> parseDefns ts = ([], ts)
```

This completes the definition of parseDefns.

THOUGHTFUL TESTING: We need to demonstrate that our parser is

working correctly. For example, we should show that the parser does not consume any tokens if the input does not start with a definition. The following runs confirm this, and show that the return result in these cases is an empty list of definitions:

```
Parse> parseDefns (lexProp "")
([],[])
Parse> parseDefns (lexProp "A")
([],[TVAR "A"])
Parse>
```

We should also check that our parser works in cases where there is a single definition in the input. It is useful to try tests that show it will accept different forms of Prop on the right hand side of a definition and not just some fixed Prop form:

```
Parse> parseDefns (lexProp "A = NOT B;")
([("A",NOT (VAR "B"))],[])
Parse> parseDefns (lexProp "A = AND A B;")
([("A",AND (VAR "A") (VAR "B"))],[])
Parse>
```

Note that all of the input tokens are consumed in these examples and that both return a singleton list corresponding the single definition in the given input strings.

If we add more elements to the input string, then the list of parsed definitions grows in the same way. This shows that our parser can handle lists with multiple definitions and that it will stop, returning unmatched tokens in the second component of the result pair, when it finds input that does not match a definition:

```
Parse> parseDefns (lexProp "A = AND A B; C = OR D E;")
([("A",AND (VAR "A") (VAR "B")),("C",OR (VAR "D") (VAR "E"))],[])
Parse> parseDefns (lexProp "A = AND A B; C = OR D E; F = G;")
([("A",AND (VAR "A") (VAR "B")),("C",OR (VAR "D") (VAR "E")),("F",VAR
"G")],[])
Parse> parseDefns (lexProp "A = AND A B; C = OR D E; AND NOT A")
([("A",AND (VAR "A") (VAR "B")),("C",OR (VAR "D") (VAR "E"))],
[TAND,TNOT,TVAR "A"])
Parse>
```

We should also check to make sure that our parser generates suitable errors if we omit a semicolon, or if we write a malformed Prop in a list of definitions:

```
Parse> parseDefns (lexProp "A = B")

Program error: missing semicolon

Parse> parseDefns (lexProp "A = NOT;")

Program error: pattern match failure: parseProp [TSEMI] ++ lexProp []

Parse>
```

The latter error message is a bit awkward; it occurs because we did not include code to handle a Token stream that begins with TSEMI in our parser for Prop. (We had not defined the TSEMI token when we wrote that parser; with hindsight, we should probably go back and add a line "parseProp ts = error ..." at the end of the parseProp definition to take care of this.)

Finally, it is usually a good idea to check that our program produces outputs that agree with the examples given in the question:

```
Parse> parseDefns (lexProp "A = AND B C; C = NOT E;")
([("A",AND (VAR "B") (VAR "C")),("C",NOT (VAR "E"))],[])
Parse> parseDefns (lexProp "A = AND B C; C = NOT E")

Program error: missing semicolon

Parse> parseDefns (lexProp "A = A; B= A; B=B; NOT VALID")
([("A",VAR "A"),("B",VAR "A"),("B",VAR "B")],[TNOT,TVAR "VALID"])
Parse>
```

By visual inspection, the first and third examples agree with the outputs shown in the question. The second example does not produce exactly the same output, in part because the text of the error message that we used is different (it does not include the "!" at the end) but also because our definition does not output the first part of the result list. However, it does detect and report the same error as the example in the question, and we consider that the correct behavior in this case.

But why is there a difference in the outputs? The version of parseDefns that was used to generate the output above was actually written using a slightly different Haskell language feature called a "let" expression that had not been introduced at the time this assignment was set (but it is introduced in the Week 4 labs). As

such, students were not expected to be able to reproduce exactly the behaviour shown in the question. But if you did not try the tests given in the question, or if you did not notice a discrepancy in the output results, then you may not have been paying careful enough attention to your testing process!

[The remaining details are beyond the scope of this course; you will not be expected to know about the following in an exam or future homework assignment, but may still find some useful/interesting ideas here. Please ask if you would like to know more about this and I will be happy to discuss that with you!]

For completeness, the version of the code using a "let" expression was as follows:

```
  parseDefns :: [Token] -> ([(String, Prop)] , [Token])
  parseDefns (TVAR v : TEQ : ts)
    = case parseProp ts of
        (p, TSEMI : ts1) -> let (ds, ts2) = parseDefns ts1   -- ***
                            in ((v, p) : ds, ts2)            -- ***
        (p, ts1)         -> error "missing semicolon"
```

The change here is marked by the *** symbols, showing where we have used a let expression to capture the results produced by the recursive call to parseDefns instead of the case expression in the original version. This allows the "lazy evaluation" feature of Haskell to be used, allowing the interpreter to delay the recursive call until the value of ds is actually needed ... in this case, that only occurs after we have output the (v, p) pair for the first definition in the input.
