

Schedule: The **Final Exam** is on **Tuesday, May 4, 11:45am–2:45pm** in **Hochstetter 114**. It will be *open-book, open-notes*. It will cover all of the course material, Chapters 1–5 and 7, skipping the regular-languages Pumping Lemma and the Post Correspondence problem, but parts of Section 9.3 needed for Theorem 9.34 (the “alternative proof of the Cook-Levin theorem,” 2nd. ed. only?). There will be one more problem set, due on the last day of class, which is *Monday, April 26*. I intend to have an optional pre-exam “Review Session” on *Monday, May 3*.

Reading: Next Monday’s lecture will begin with the definition of mapping reducibility in section 5.3, then do more examples there and in Section 5.1. For Wednesday and the last Monday, read Chapter 7 and Section 9.3. Section 5.2 and all of Chapter 6 are skipped, while lecture will replace the long proof of the Cook-Levin Theorem in section 7.4 with the shorter proof in section 9.3.

(1) Text, problem 4.26 on page 184: Sketch in prose a decision procedure for the following problem:

Instance: A CFG $G = (V, \Sigma, \mathcal{R}, S)$, and a string $x \in \Sigma^*$.

Question: Does G generate some string $w \in \Sigma^*$, such that x is a substring of w ?

You may use the fact that since x is fixed, the language $\Sigma^*x\Sigma^*$ is regular. Then you may use as a step in your sketch the fact that given G and any regular set R , a computable function can build a context-free grammar G_R such that $L(G_R) = L(G) \cap R$. The text also hints about appealing to the decision procedure in Theorem 4.8 for E_{CFG} . (15 pts.)

(2) Prove that it is undecidable whether a given coffee-maker will ever brew a cup of coffee. The original purpose of Sun’s Java programming language project, before the WWW applications burst on their consciousness, was a small-kernel universal platform for “Embedded Systems,” i.e. uniform ways of programming controls for household appliances, TVs, and the like. So suppose we’re talking about coffee-makers with full JDK 2.0 installed and access to unlimited storage via broadband. The manufacturer soft-wires a control program P , which can be any legal Java program, into the “Java chip” in the maker—and on high-end models, users can download patches and updates into their makers via the wireless broadband connection. There is room in most ROMs to fit the *Turing Kit* program TK as well, and combine it with P into an overall program Q that can call TK as a subroutine—and you can have TK access any fixed Turing machine M by putting the file for M in your CSE396 directory and letting your coffee-maker log in to it. *Hint:* If M accepts its own code, brew coffee.

Now use this idea seriously to show that the following problems are undecidable:

- (a) Given a Java program P and a line of code ℓ , does there exist an input x to P that causes line ℓ to be executed? (If not, then we can safely delete line ℓ —this problem is interesting to those trying to reduce “code bloat.” 12 pts.)
- (b) Given a Java program P , is there an input x to P that causes P to raise an `ArrayIndexOutOfBoundsException`? Here it may help you to know that the *Turing Kit* program, though it has other bugs, never raises this exception. (12 pts.)

Note: To handle problems of the “is there an input x ” type, i.e. where only a program is given, you may find the “ignore-input” idea of the reduction from D to E_{TM} in class useful. The “moral” is that almost any particular behavior of programs—unless that behavior is required to simulate Turing machines i.e. be already part of TK —is impossible to predict and analyze in all cases. (*Self-study problem, not for credit: do Problems 5.14 and 5.15 on p212 (1st ed., p195) with these remarks in mind.*)

(3) Text, exercise 5.12 on page 211 (1st ed., on page 195). Although couched in Turing machine terms, this is really “the same” basic idea as in problem (2). 18 pts., for 57 total not counting the extra-credit option next.)

(4) **Extra Credit:** The motivation for this problem is that although Sipser’s text in section 7.2 proves that every individual context-free language belongs to the class P , it does not prove that the Acceptance Problem for CFGs, standardly denoted by A_{CFG} , belongs to P . The reason is that it requires the given grammar G to be in Chomsky normal form. An instance of the A_{CFG} problem has the form $\langle G_1, x \rangle$ where G_1 can be a CFG in **any** form, and the gap is that the standard way of describing the conversion to Chomsky normal form can take time exponential in the size of G_1 , just to deal with the ϵ -rules.

To overcome the gap and do something even more significant, give an algorithm that converts an EBNF grammar G_0 into an ordinary CFG G_1 **in polynomial time**. Note that the EBNF-to-CFG conversion described in lecture does not suffice: it would translate an EBNF rule

$$A ::= [B_1][B_2][B_3]\dots[B_n]$$

into 2^n CFG rules, each with a different combination of the B_i ’s. Instead, recalling that the final step of conversion to Chomsky normal form is to shorten right-hand sides to have two concatenated *variables*, shorten cases like this to have two concatenated *components*, using extra variables to connect them:

$$A ::= [B_1]R_1, \quad R_1 ::= [B_2]R_2, \quad \dots \quad R_{n-1} ::= [B_{n-1}][B_n].$$

Then removing the $[\dots]$ produces only $2(n-2)+4$ rules, at the cost of $n-1$ new variables, but the total size blowup stays linear for that rule and that step of the conversion. Of course you also need to handle internal ORs (i.e., $|$ in BNF notation, $== +$ or \cup in regexp notation) and braces $\{\dots\}$ (which $==$ Kleene stars $*$ in regexp notation), but the idea of substituting a list-making variable for a brace works fine, and the internal ORs don’t “blow you up exponentially” if you first handle multiple concatenations as above. For full credit your algorithm must work by recursion/induction on the formulas by which right-hand sides are built up out of EBNF operations, and you must maintain the runtime analysis correspondingly.

Now your G_1 itself need not be in Chomsky NF, but you can *re-use* the above idea to handle the removal of ϵ -rules, aided by the fact that the algorithm to determine which variables are capable of deriving ϵ runs in polynomial time. That algorithm is:

```
Set<Var> EVARS = Set<Var>.Emptyset; bool changed = true;
while(changed) {
    changed = false;
    foreach(rule A --> X_1X_2...X_m where A \notin EVARS) {
        if (each X_i is in EVARS) {
            EVARS += {A}; changed = true;
        }
    }
}
return EVARS;
```

Each iteration through the rules either adds a new variable to **EVARS** or halts the whole thing with no change, so the running time is order-of the number of rules times the number of variables, which is polynomial in the size of G_1 . Also note that G derives ϵ if and only if the final **EVARS** contains the start symbol S , so the problem of deciding whether a given CFG generates ϵ is in P . (And, if you initialize **Set<char> LIVE = \Sigma** in place of **EVARS**, it is identical to the algorithm at the very end of today’s lecture for deciding whether $L(G) \neq \emptyset$.) Put this all together with the text’s dynamic-programming algorithm in section 7.2 (which works only for nonempty x , and which I won’t cover) to conclude that A_{CFG} is polynomial-time decidable. (24+12 = 36 points).