Lecture 1-Detecting Correct Placement of Brackets

1 The Problem

A fundamental problem in Computer Science is to parse and evaluate complicated algebraic expressions, like

\[(3 \times (x + 8) - 16 \times (7 + (y - 3)/z))/(5 + 6 \times (x + 4))\].

If you forget about the operators and operands in the expression, and just write down the parentheses, you wind up with:

\[(()))(()())\].

Any expression-evaluator has to be able to determine, among many other things, whether these parentheses are correctly nested. (In this case they are.) Likewise, a compiler for a program written in a language like Java or C has to determine if the occurrences of the braces

\n
\{, \}

are correctly nested. That is the problem we will consider here. The most desirable end result is an efficient algorithm that takes a string of parentheses as input and outputs “yes” or “no”, depending on whether the string is correctly formed.

2 Aside–A Reduction

What we have done is taken a problem (parsing and evaluating expressions, or programs) and decided to consider a simpler problem—that of testing
for correct parentheses. This is a typical approach in all kinds of problem-solving: If we want to be able to solve the harder problem, then surely we will have to know how to solve the simpler one.

Here is one way to solve our problem. Take a string of braces, and prepend to it the line:

```java
global class Empty
```

and then save the result in a file called `Empty.java`. Then run the Java compiler on this file. If the compiler exits without printing an error message, then the string was correctly formed; otherwise not.

Now this “solution” is both eminently practical (you have the Java compiler handy, so you can apply this method right away) and ridiculous (because it uses such complicated machinery to solve a simple problem). It doesn’t really solve the problem in the sense we asked for, since the actual algorithm for parenthesis-testing is hidden inside the compiler. Nevertheless, it illustrates an important principle that we will emphasize throughout the course: We have described a reduction of the problem at hand (testing for correct bracketing) to another problem (compilation): From a program that solves the second problem, we create a program that solves the first (we could carry out the process of prepending the additional text and intercepting the error message automatically). The reduction proves in a precise sense that our problem is simpler than that of compilation.

3 A Typographical Simplification

It’s a bit awkward to write strings of parentheses like the ones above, so let’s just write `a` for a left parenthesis and `b` for a right parenthesis. So now our program is supposed to take inputs like

```
aabaabbaabb
```

and answer yes or no. What’s the answer in this case? How do you know?

4 Some Proposed Solutions

I have posed this problem to many classes of students. The first solution proposed is usually:
The numbers of left parentheses and right parentheses have to be equal.

But the student who proposes this will almost immediately correct himself or herself and realize that this condition, while certainly necessary, is not sufficient, as the example 

\[ ba \]

shows. Let’s try again:

The numbers of left parentheses and right parentheses have to be equal, the first symbol must be a left parenthesis, and the last symbol must be a right parenthesis.

But this is still not sufficient—look at

\[ abbaab. \]

Sooner or later someone tries

Every left parenthesis has to have a matching right parenthesis,

The problem here is one of vagueness. What does it mean for a right parenthesis to match a left parenthesis? If it just means some pairing that associates to every left parenthesis in the string a unique right parenthesis, and vice-versa, then this is no different from the requirement that the number of left parentheses be equal to the number of right parentheses, and we’ve already seen that this doesn’t work.

Think about the problem a little harder, and try to formulate a correct criterion before reading further.

5 Definition of Correct Bracketing

One difficulty is that we haven’t really said what a correct parenthesization is. Somehow there’s this feeling that you know one when you see one, and you know an incorrect one when you see one. That is enough to prove that the first two criteria proposed above are insufficient. But it won’t do it all if we ever find the right answer and have to prove that it works.

To formulate a good definition for correct parenthesizations, let’s consider how algebraic expressions are built. We start with atomic expressions,
which are just numbers and variables. We make larger expressions out of two smaller ones by putting an operator (like + or ×) between them. And we can make a larger expression out of a smaller one by putting a pair of parentheses around it.

If you look at the above description and ignore everything but the parentheses, you get the following recipe for generating correct sequences of parentheses:

- The empty string of parentheses is correct.
- If you concatenate (join together) two correct strings, you get a correct string.
- If you put parentheses around a correct string, then you get a correct string.
- All correct strings are formed in this way.

We spend a lot of time in this course talking about strings or words over an alphabet of symbols or letters, so it is helpful to have some precise notation and terminology for talking about strings. Precise mathematical notation can be terribly cryptic-looking before you master the language, and sometimes even after you do. But it is far more concise and unambiguous than trying to write everything out in ordinary English.

In the case we consider here, the alphabet is the set \{a, b\}. We denote the set of all strings over this alphabet by \{a, b\}*. The set \{a, b\}* contains an empty string, which we denote by \(\epsilon\).

If \(u\) and \(v\) are strings, then \(uv\) denotes the string that results by concatenating the two together. We also sometimes write \(u \cdot v\). We write \(|u|\) to denote the length of the string \(u\). For instance

\[|aababab| = 7,\]

and

\[|\epsilon| = 0.\]

Obviously,

\[|uv| = |u| + |v|\]

for any two strings \(u\) and \(v\).

We now give the “official” definition: The set \(P \subseteq \{a, b\}^*\) of correct parenthesizations: satisfies
• $\epsilon \in P$;
• if $u, v \in P$ then $uv \in P$;
• if $u \in P$ then $aub \in P$;
• all elements of $P$ are obtained by repeated application of the three preceding properties.

The fourth item in the definition makes this a recursive definition. Here is another way to write the definition that makes the recursion more like the recursive methods you’re accustomed to seeing in programming classes:

A string $w \in \{a, b\}^*$ belongs to $P$ if and only if either (a) $w = \epsilon$; (b) $w = uv$, where $u \in P$ and $v \in P$ and $|u| < |w|$; (c) $w = aub$, where $u \in P$.

Recursion is sometimes confused with circularity: We appear to be defining a property (correct bracketing) in terms of itself, and thus are not defining anything at all. But what we are really doing is defining the property in terms of smaller instances of itself, and that makes all the difference.

6 An Algorithm for Testing Correct Bracketing

Our definition gives a method for generating correct strings of parentheses, but what we asked for was a method of testing a string of parentheses to see if it’s correct. In fact, our recursive definition for generation can be turned directly into a recursive Java method for testing, but the resulting algorithm turns out to be very unsatisfactory in practice. (This is explored in the exercises.)

You get a better algorithm if you closely scrutinize what is wrong about the second proposed solution given above:

$$abbaab.$$ 

If you scan this string from left to right, you realize that something is wrong as soon as you hit the second occurrence of $b$. In the vague terms we criticized above, there is no matching $a$ for this $b$. A simpler and more precise way to formulate what is wrong is that in a left-to-right scan, the number of $a$‘s encountered always has to be at least as large as the number of $b$‘s; that is,
If \( w \in P \) and \( w = uv \) for some strings \( u \) and \( v \), then the number of \( b \) in \( u \) cannot be more than the number of \( a \) in \( u \).

This section promises an \textit{algorithm} for testing correct bracketing, but what we wrote looks more like another mathematical definition than a computer algorithm. That’s easily remedied—here is a pseudocode version of the same criterion:

```cpp
excess = 0;
while more symbols remain {
    if next symbol is a
        excess = excess + 1;
    else if next symbol is b
        excess = excess - 1;
    if (excess == -1)
        reject and exit;
}
if (excess == 0)
    accept;
else
    reject;
```

Is this enough now? It seems reasonably clear that any correct bracketing has to satisfy this criterion, but can we rule out the possibility of some weird counterexample that passes this test, and is nonetheless not correctly bracketed?

7 A Proof that the Criterion is Correct

We can if we manage to \textit{prove} that the strings of symbols accepted by this algorithm are exactly the ones that belong to \( P \), according to the definition above. Let’s first introduce a little additional notation. It is awkward to continually have to say “the number of \( a \)’s in \( u \)”, so let’s denote this number by \( |u|_a \).

So we want to prove the following theorem:
**Theorem 1** Let \( w \in \{a, b\}^* \). \( w \in P \) if and only if \(|w|_a = |w|_b\), and whenever \( w = uv, |u|_a \geq |u|_b\).

### 7.1 Proof Strategy

A formal proof can be difficult to follow and obscure what is often a very simple idea. So before presenting the official argument, let’s first present the argument informally.

The “if and only if” means that there are two things to prove—that a string in \( P \) satisfies these properties, and that every string that satisfies these properties is in \( P \).

There is a way to literally see the argument in both directions. Let’s associate to each string \( w \) in \( \{a, b\}^* \) a graph whose height above the \( x \)-axis at \( x = k \) is the excess of \( a \)'s over \( b \)'s in the first \( k \) letters of \( w \). The domain is the set \( \{0, 1, \ldots, |w|\} \).

![Graph of u=abababbb](image)

![Graph of v=ababababbb](image)
The two properties in the statement of the theorem can then be rephrased: The graph of \( w \) ends on the \( x \)-axis, and the graph of \( w \) never crosses the \( x \)-axis.

Now it really is obvious that if \( u \) and \( v \) are strings whose graphs satisfy the above properties, then the same is true of both \( uv \) and \(aub\), so as we apply the rules to build longer and longer strings in \( P \), the properties in the theorem are always preserved. We thus conclude that every string in \( P \) satisfies these properties.

You should also note that the graph of \( auv \) never touches the \( x \)-axis, except at the very beginning and the very end.

The other thing to prove is that if a string satisfies the two properties, then it is in \( P \). We can basically reverse the argument above: If \( w \) is a nonempty string that satisfies the two properties, then its graph either touches the \( x \)-axis somewhere in the middle, or it does not. If it does touch, then it is formed by concatenating two graphs, both of which satisfy the required properties.
Thus $w = uv$, where $u$ and $v$ are shorter strings satisfying the properties in the theorem, so if the result is true for these shorter strings (implying $uv \in P$), then it must be true for $w$ (implying $w \in P$). Alternatively, if the graph of $w$ never touches the $x$-axis, then the portion of the graph between $x = 1$ and $x = |w| - 1$ starts and ends at $y = 1$ and never falls below the line $y = 1$. Thus if we lower this portion 1 unit, we obtain the graph of a string $u$ that also satisfies the properties. Hence $w = aub$, where $u$ satisfies the properties in the theorem. Again, if the theorem holds for the shorter string $u$, then $u \in P$, and hence $w = aub \in P$.

### 7.2 The Formal Proof

**Warning:** This part is hard to read. Don’t worry too much if you don’t get it all at first, but you should definitely make the effort to slog through it and see how it mirrors the pictorial argument given above. In both directions of the informal proof, we inferred the truth of the theorem for longer strings from its truth for shorter strings. This means that we are giving a proof by induction on the length of a string.

We make this precise, first with the proof that every string in $P$ satisfies the properties of the theorem.

Let $w \in P$. To start the induction, we first consider the case $|w| = 0$, so that $w = \epsilon$. This is indeed in $P$, so we have to verify that it satisfies the above properties, which it clearly does, since $|\epsilon|_a = |\epsilon|_b = 0$, and since the only possible factorization of $\epsilon$ is $\epsilon = \epsilon \cdot \epsilon$.

Now suppose $|w| = N > 0$. The inductive hypothesis is that all strings in $P$ of length less than $N$ satisfy the two properties. Now either $w = xy$, where $x, y \in P$ and $|x|, |y| < N$, or $w = axb$, where $x \in P$.

In the first case, $|w|_a = |x|_a + |y|_a = |x|_b + |y|_b = |w|_b$. Note that the second equality follows from the inductive hypothesis. Further, if $w = uv$, then either $u$ is a prefix of $x$, and thus $|u|_a \geq |u|_b$ by the inductive hypothesis; or $u = xy'$, where $y'$ is a prefix of $y$, and thus

$$|u|_a = |x|_a + |y'|_a = |x|_b + |y'|_a \geq |x|_b + |y'|_b = |u|_b.$$ 

Again, the second equality and the inequality above both follow from the inductive hypothesis.

In the second case, where $w = axb$, we have $|w|_a = 1 + |x|_a = 1 + |x|_b = |w|_b$, by the inductive hypothesis. If $w = uv$ then either $u$ is empty, in which
case $|u|_a = 0 = |u|_b$, or $u = ax'$, where $x'$ is a prefix of $x$. In this case, the inductive hypothesis gives us

$$|u|_a = 1 + |x'|_a > |x'|_a \geq |x'|_b = |u|_b.$$  

That proves the ‘only if’ part. Now we have to show that any string $w$ satisfying the conditions in the theorem belongs to $P$. Again, the proof is by induction on $|w|$. If $|w| = 0$, then $w = \epsilon$. This satisfies the two conditions and does indeed belong to $P$. Now suppose $|w| = N > 0$ and that $w$ satisfies the conditions in the theorem. The inductive hypothesis is that any string of length less than $N$ satisfying these conditions is in $P$.

There are two possibilities for $w$. Either there is some proper prefix $x$ in which the number of occurrences of $a$ is equal to the number of occurrences of $b$—that is, $w = xy$ with $|x| < |w|$ and $|x|_a = |x|_b$, or not. If such an $x$ exists, then every prefix $u$ of $x$ is a prefix of $w$, and thus $|u|_a \geq |u|_b$. Further, if $u$ is a prefix of $y$, then $xu$ is a prefix of $w$, so

$$|xu|_a \geq |xu|_b.$$  

Thus

$$|u|_a = |xu|_a - |x|_a \geq |xu|_b - |x|_b = |u|_b.$$  

It follows from the inductive hypothesis that both $x$ and $y$ are in $P$, so by the definition, $w = xy$ is in $P$.

If no such prefix $x$ exists, then the first letter of $w$ must be $a$, otherwise we would violate the condition on prefixes of $w$. The last letter of $w$ must be $b$, otherwise $w = ya$ and $y$ would have more occurrences of $b$ than $a$. So $w = axb$ for some $x$. We want to show $x \in P$, which will imply $w \in P$. Consider a prefix $u$ of $x$. We know $|au|_a \geq |au|_b$, since $w$ satisfies the conditions of the theorem, and further we know $|au|_a > |au|_b$, since we supposed that $w$ has no prefix with the same number of $a$’s and $b$’s. Thus $|u|_a \geq |u|_b$. Also $|x|_a = |w|_a - 1 = |w|_b - 1 = |x|_b$. Thus by the inductive hypothesis, $x \in P$, so $w = axb \in P$. We’re done!

That took a long time to say in precise mathematical language, and it’s easy to get lost. But it’s really just a formalization of the argument with the graphs given above.
8 Recapitulation

Some important ideas, which we will see throughout the course, occur in the notes above. Here is a summary:

- **Reduction.** Assuming we have an algorithm for Problem B, we can construct an algorithm for Problem A—that is, we have reduced Problem A (parenthesis-testing) to Problem B (compilation). Here the reduction was very simple, hardly more than a single call to the algorithm for B.

- **Problems as Decision Problems about Sets of Strings.** Parenthesis-testing is explicitly a problem about testing whether an input string belongs to a certain set $P$ of strings. But the input to any computational problem can be encoded as a string of symbols, and many—although not all—problems have yes-no answers as output. We will see such problems time and again....

- **Special Notation and Terminology for Strings of Symbols** ...which is why we require special notation for strings of symbols. Learn it! You’ve already studied the standard notation for sets and functions, but just in case it got past you, learn it!

- **The Need for Formal Definitions and Proofs.** We developed a method for testing whether a string of parentheses is properly formed. But how do we know that our method is correct? Other reasonable criteria that we saw turned out to be wrong. The only way we can be sure is to develop a precise definition of what constitutes a correct string of parentheses, and to prove that our method accepts all and only those strings.

- **Recursion and Induction** The definition of a correct string is recursive, so it is only natural that the proof of a theorem about correct strings is by induction. Recursion and induction go hand in hand, and it is important that you understand how they work. Do not, however, be misled into thinking that every proof is a proof by induction.
9 Problems

1. Implement the algorithm for parenthesis-testing described in Section 6. If you write in Java, give the string input as a command-line argument. So you would type something like

```
java Test (())(()(()(1)
```

to run the program, and the program will respond (in this case) ‘No’.

2. How much memory does your program in the previous problem use? There is, of course, the memory required to hold the input string, but how much additional memory is used? Does it depend on the length of the string?

3. Here is a way to state the vague criterion ‘there is a right parenthesis for every left parenthesis’ in a precise fashion: There is a one-to-one function \( f \) from the set of left parentheses onto the set of right parentheses in the string such that for every left parenthesis \( x \), \( f(x) \) is to the right of \( x \). (The term one-to-one function is not defined until page 175 of the text, in Chapter 4, but it could have, and should have, been defined in Chapter 0.) Again, it’s pretty obvious that this criterion is necessary. Is it sufficient?

   (a) Give an example to show that if such a one-to-one function exists, it need not be unique. Thus it is possible to match a left parenthesis to the wrong right parenthesis.

   (b) Prove that, nonetheless, this criterion is indeed sufficient. (Don’t try a fancy proof by induction. Instead give a simple direct argument that this new criterion implies the properties in Theorem 1.)

   (c) Unlike the properties we carefully developed in the lecture, this one does not seem to result in a nice algorithm for testing correct parenthesization. Why not?

4. Recursive definitions, such as the one we gave for the set \( P \), can usually be immediately converted into programs that use recursion. We thus get the following alternative algorithm for parenthesis testing. (In this
pseudocode, execution of the return statement causes the program to terminate.)

if string is empty
    return true;
if first symbol of input is not ‘a’ or last symbol is not ‘b’,
    return false;
if substring from second up to next-to-last symbol is correct
    return true;
for i = 1 to (length of string)-1
    if substring from 1 to i-1 is correct
        if substring from i to (length of string)-1 is correct
            return true;
    return false;

(a) Implement this algorithm and test it on a few strings.
(b) Comment on the efficiency of this algorithm. If you know how
to, set up and solve a recurrence relation giving the worst-case
running time of this algorithm on strings of length \( n \).