MAU11602
Lecture 20
2026-03-12

Balanced parentheses example

Does `(()(())(()))()(())` have balanced parentheses?

```
   (  (  )  (  (  )  )  (  (  )  )  )  (  )  (  (  )  )
0  1  2  1  2  3  2  1  2  3  2  1  0  1  0  1  2  1  0
```

so yes.

Can we really be sure though?

- The empty list has balanced parentheses.
- If we take a list with balanced parentheses and put a (before the start and a) after the end then the resulting list has balanced parentheses.
- Appending one list with balanced parentheses to another list with balanced parentheses gives a list with balanced parentheses.

It follows that every list which is grammatically correct according to the grammar

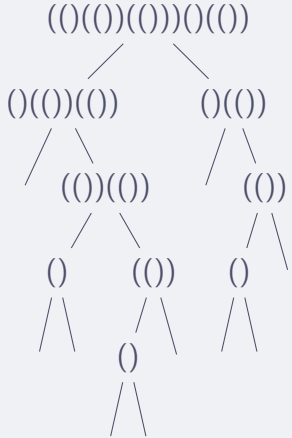
```
ok ::= | "(" ok ")" ok
```

has balanced parentheses.

The converse is less obvious, but still true.

Balanced parentheses example, continued

This is the parse tree for `(()(()(()))()(()))`, showing all subexpressions.



It's easy to check that all leaves have the empty expression and all other nodes have a `(`, then the left child's value, then a `)`, then the right child's value.

Balanced parentheses example, conclusion

If you want to do less work and make your reader do more work there's another to convince them that `(()(())(()))()(())` has balanced parentheses.
Consider the string
`,(),(()),()(()),(())(()),()(())(()),(()(())(()))()(())`
This is a list of subexpressions, starting with the empty one and ending with whole expression.
It's ordered in such a way that the correctness of each expression, i.e. the fact that it has balanced parentheses, follows from the correctness of ones earlier in the list.
For the first expression this is easy, and for all the other ones you can check that they are a `(`, then some expression earlier in the list, then a `)`, then some "other" expression earlier in the list.
This is annoying to check, but a much more concise representation of our proof that the list satisfies the grammar.
It doesn't belong to the language of balanced parentheses, but to another language, with three symbols, `(`, `)`, and `,`, in its alphabet.

Embedding the old language in the new

The new language, with `,`, also includes all lists with just `(`'s and `)`'s.
We can represent both with modified base 3. For definiteness, I'll take the digits 1, 2, and 3, to represent the symbols `(`, `)`, and `,`, in that order.
For example, `(()(())(()))()(())` is represented by 112112211222121122, the modified base 3 representation of the natural number 208777301.
There are no 3's in the representation, reflecting the fact that there are no `,`'s in the list. Equivalently, it is not possible to write this list as the concatenation of three lists, the middle one of which is the list `,`, of length 1.
Let $\alpha(p, a, b, c)$ be the expression, we found earlier which expresses c representing the concatenations of the lists representing a and b in the modified base p representation. Then the absence of `,`'s in the list represented by a is

$$\neg\exists b.\exists c.\exists d.\alpha(3, b, 3, d) \wedge \alpha(3, d, c, a).$$

So the set of numbers representing lists without a `,` is arithmetic.

Correctness

So the lists of just (’s and)’s form an arithmetic set. What about those which also have balanced parentheses?

Those are precisely the ones for which there exists proof like the ,(),(()),()(()),(())(()),()(())(()),((()(())(()))()(()) we saw earlier.

What do we need to verify, and can it be expressed in terms of list concatenation?

In other words, if a is the natural number representation of the list of parentheses we want want to check for balance and b is number representing a purported proof then can we express the validity of the proof as a Boolean expression in Peano arithmetic with free variables a and b, using the expression α we already have?

One obvious condition is that unless $a = 0$ the list for b should be something, followed by a ,, followed by the list for a.

In Peano arithmetic,

$$\exists c.\exists d.\alpha(3, c, 3, d) \wedge \alpha(3, d, a, b).$$

This says the purported proof ends with the statement we’re trying to prove, which is necessary, but not sufficient.

Correctness, continued

Besides checking that the proof ends correctly, we need to make sure each step is justifiable.

In other words, if a list of parentheses appears then either it is empty or two other lists appear before it such that the result of concatenating `(`, one of the lists, `)`, and the other list is the given list.

All of these conditions are expressible. For example, Saying the a appears before b in c is the same as saying the c is the concatenation of d, a, `,`, e, b, and f, where d is empty or the concatenation of some g and `,`, e is either empty or the concatenation of some h and `,`, and f is either empty or the concatenation of `,` and some i.

In Peano arithmetic, with the α shorthand and n being the digit corresponding to the symbol `,`, this is

$$\exists d.\exists e.\exists f.\exists g.\exists h.\exists i.\exists j.\exists k.\exists l.\exists m.\alpha(p,d,a,j) \wedge \alpha(p,j,n,k) \wedge \alpha(p,k,e,l) \wedge \alpha(p,l,b,m)$$
$$\wedge \alpha(p,m,f,c) \wedge (d = 0 \vee \alpha(p,g,n,d)) \wedge (e = 0 \vee \alpha(p,h,n,d)) \wedge (f = 0 \vee \alpha(p,n,i,d)).$$

From now on I'm going to stop writing expressions explicitly and just say that an appropriate expression exists.

Correctness, concluded

We could do something similar for the condition that b appears in c, or we could just note that that's the same as saying b is empty or the empty list appears before b in c, which we already know how to express.

Now we have a bunch of expressions which express all of the necessary conditions for a list b in our extended language to be a proof that a given list a of parentheses is balanced. We can just $\wedge$ them together and stick a $\exists b$ in front to get a statement saying that a has balanced parentheses.

So the set of encodings of lists with balanced parentheses is arithmetic.

Generalisations

The details of the preceding slides depend on the details of the grammar

```
ok ::= | "(" ok ")" ok
```

but the method applies to any grammar.

Given a grammar for a language based on an alphabet with k symbols we can find a prime $p > k$ and then an expression with one free variable in Peano arithmetic saying the list encoded by the number that variable stands for belongs to the language with that grammar.

All formal languages, sets of lists of symbols parsable with a grammar, are arithmetic!

We can push this idea further though. If we can implement parsing in Peano arithmetic then what about lexing, type checking, or evaluation?

All of these work! Any operation on lists of symbols implemented by a sequence of steps and with rules for the allowed steps expressible in terms of concatenation can be represented in Peano arithmetic.

We generally need to add extra symbols to our language and choose a higher value of p, but we can always make a single choice which works for everything we want to do.

Programming in Peano arithmetic

Choosing a programming language and a sufficiently large prime p we can find an expression β such that $\beta(a, b, c)$ is true precisely when the a is the representation of a function in the language, b and c are representations of values in the language, and the function represented by a applied to the value represented by b evaluates to the value represented by c.

There is one subtle point. I've been treating lists of symbols and natural numbers as essentially the same thing. Most programming languages allow us to represent natural numbers in the language as some list of symbols. Our encoding converts this list of symbols to an integer, but not generally the integer we started with.

I'm going to assume that my language has a notation for zero and an increment function so we can represent natural numbers in the language by repeatedly incrementing zero.

Let s be the encoding of the increment function and let o be the encoding of the zero value. Let $g(n)$ be the encoding of the natural number n, obtained by repeated incrementing, so $g(0) = o$ and $\alpha(s, g(n), c)$ if and only if $c = g(n + 1)$.

Programming in Peano arithmetic

Suppose for the moment that the function g is arithmetic, and a function f is definable in our programming language.
In other words, there is some expression γ in Peano arithmetic such that $\gamma(m, n)$ is true if and only if $g(m) = n$.
Let

$$\varphi(i, j) = \exists a.\exists b.\gamma(i, a) \wedge \gamma(j, b) \wedge \alpha(c, a, b)$$

where c is the encoding of f.
Then $\varphi(i, j)$ is true if and only if $f(i) = j$.
So f is an arithmetic function.
In other words, anything computable is arithmetic!
Getting here was painful, but now we can dispense with all the ad hoc arithmeticity arguments we've been using.
Or rather we will be able to once we show g is arithmetic.

Encoding is arithmetic

If we already knew that everything computable was arithmetic then it would be easy to show that encoding is arithmetic, but that argument is circular, so we need a different one.

The basic idea is the one we used for showing that parsing is arithmetic: we construct evidence that the statement is true and then define our relation by the existence of such evidence.

The evidence has to be checkable using only logic and concatenation.

The obvious evidence for $g(m) = n$ is the sequence of ordered pairs $(0, g(0)), (1, g(1)), \ldots, (m, g(m))$.

We can check whether a sequence of ordered pairs $(a_0, b_0), (a_1, b_1), \ldots, (a_k, b_k)$ is proper evidence by checking if

- the initial pair is $(0, o)$,
- the final pair is (m, n), and
- for every to adjacent pairs (a, b) and (c, d) we have $c = Sa \wedge \beta(s, b, d)$.

Encoding is arithmetic, continued

We might hope to extend our language with `(`, `,`, and `)` to cope with lists of ordered pairs.
Or some new separator and pair of delimiters if those are already in our language.
Unfortunately this is the one place where extending the alphabet is not permitted.
We can get around this though. First of all, we only really need new delimiters, since we can extract everything we need from a representation like $[[a_0][b_0]][[a_1][b_1]] \cdots [[a_k][b_k]]$.
Second, instead of using new symbols, we can use a sequence of old symbols, provided they don't occur elsewhere in our list of symbols.
Unfortunately every possible sequence occurs in the evidence of for $\gamma(m, n)$ for some m and n, so we need to use a different pair for each proof.
This is okay. We just need to start our proof with a header identifying which sequences will be used as delimiters.
Now, finally, we can say that everything computable is arithmetic.