CS 241E Final Review

Fall 2019

1 Closures

Consider the partially-implemented Lacs program below.

```
def main(a:Int,b:Int):Int = {
    var pair:(Int)=>Int;
    pair = newPair(a,b);
    pair(2);
    pair(0) - pair(1)
}

// Creates a pair of integers.
// This is represented as a closure which takes one argument: an integer
// representing a choice of method to perform on the pair.
// The following methods are available:
// 0    (getFirst): Return the first integer.
// 1   (getSecond): Return the second integer.
// 2 (compareSwap): If the first integer is smaller than the second integer,
//                  swap them. Return 1 if a swap occurred and 0 otherwise.
def newPair(a:Int, b:Int):(Int)=>Int = {
    ???
    def    getFirst():Int = { ??? }
    def   getSecond():Int = { ??? }
    def compareSwap():Int = { ??? }
    def selectMethod(m:Int):Int = {
        if(m==0) {
            getFirst()
        } else {
            if(m==1) {
                getSecond()
            } else {
                if(m==2) {
                    compareSwap()
                } else {
                    1/0
                }
            }
        }
    }
    ???
}
```

1. What common mathematical function does `main` implement? (Hint: it's not plain subtraction.)

   Solution: it computes the distance between two integers (the absolute value $|a-b|$) by subtracting the smaller integer from the larger one.

2. Complete the implementation of `newPair` by filling in the question marks.

   Solution: Here is one possible implementation. You can also modify the parameters directly instead of using local variables. However, note that modifying parameters is only allowed in Lacs, not in Scala.

```
def newPair(a:Int, b:Int):(Int)=>Int = {
    var first:Int;
    var second:Int;
    def    getFirst():Int = { first }
    def   getSecond():Int = { second }
    def compareSwap():Int = {
        var temp:Int;
        if(first < second) {
            temp = first;
            first = second;
            second = temp;
            1
        } else {
            0
        }
    }
    def selectMethod(m:Int):Int = { ... }
    first = a;
    second = b;
    selectMethod
}
```

3. For each of the following procedures, state whether its frame and parameter chunk are stored on the stack or the heap, and explain why.

   - `main`
   - `newPair`
   - `selectMethod`
   - `compareSwap`

   Solution: A procedure's frame and parameter chunk go on the heap if and only if the procedure is ever made into a closure, or (recursively) it is the outer procedure of a procedure whose frame and parameter chunk go on the heap.

   In my implementation of the program, the only procedure that is made into a closure is `selectMethod`. Therefore:

   - `main` goes on the stack.
   - `newPair` goes on the heap since it is an outer procedure of `selectMethod`.
   - `selectMethod` goes on the heap since it is made into a closure.
   - `compareSwap` goes on the stack.

4. When the line `pair = newPair(a,b)` executes, the memory address of a chunk gets stored in the variable `pair`. What does this chunk contain?

   Solution: This chunk contains:

   - The address of the code of the procedure `selectMethod`.
   - The address of the frame of `newPair`, which is the environment for the closure of `selectMethod`.

5. When the closure call `pair(2)` happens, which procedure do we actually jump to? How does `main` know which procedure to jump to?

   Solution: We jump to the procedure `selectMethod`. We know to jump there because `pair` points to a chunk containing the address of the procedure to jump to.

6. When the closure call `pair(2)` happens, which procedure's frame does `main` pass in as a static link? How does `main` know to pass in this particular static link?

   Solution: We pass in the frame of `newPair` that was allocated during the call `newPair(a,b)`. We know to pass in this frame because `pair` points to a chunk containing the environment of the closure, which is used as the static link when the closure is called.

7. When `compareSwap` swaps the two elements of the pair, it modifies variables in some chunk of memory.

   - Why was this chunk created? Is it just a chunk to store closure information, or is it the frame or parameter chunk of a procedure?

     Solution: The chunk is either a frame or parameter chunk of `newPair`, depending on your implementation of `newPair`. In my implementation, it would be a frame since I used local variables to represent the elements of the pair. If you used the parameters directly to represent the elements of the pair, it would be a parameter chunk of `newPair` that gets modified.

     The chunk containing the two elements of the pair is *not* the same as the "closure chunk" for `selectMethod`, which just contains pointers to the code and environment for the closure to enable the closure to be called.

   - When was this chunk created? What line of `main` results in this chunk being allocated?

     Solution: The line `pair = newPair(a,b)` creates this chunk. However, note that the chunk that actually gets stored in `pair` is just the "closure chunk" for `selectMethod`. The chunk that contains the elements of the pair is just created as a side effect.

   - Explain why this chunk needs to be stored on the heap rather than the stack.

     Solution: The chunk contains variables representing the two elements of the pair that was created during the call `newPair(a,b)`. If this chunk was stored on the stack, it would be deallocated when the call to `newPair(a,b)` finishes. Then the call to `compareSwap` would allocate things on the stack and possibly overwrite the elements of the pair. Because we need the elements of the pair to last beyond the initial call to `newPair`, we must allocate the chunk on the heap.

8. Draw diagrams of the stack and heap after the line `pair = newPair(a,b)` has been executed.

   - Assume the "simple heap allocator" is being used (i.e., don't worry about garbage collection).
   - For each chunk on the stack and heap, describe its purpose (e.g., closure chunk for procedure X, parameter chunk for procedure Y, frame for procedure Z).
   - List the variables in each chunk, not including temporary variables used for things like binary operations and if statements. You should include static links. You can write "standard frame variables" as shorthand for the dynamic link, saved program counter, and parameter pointer,

which appear in the frame of every procedure. You should know what values are stored in these variables and why saving these values is important.

- For integer variables, you do not need to list the value.
- For pointer variables, indicate which chunk the variable points to; if the pointer variable is the static link of a procedure with no outer procedure, indicate that it points to address zero.
- A chunk representing a closure contains two variables: `closureCode` containing the address of the code to be executed, and `closureEnvironment` containing the environment. For `closureCode`, indicate which procedure's address is stored in the variable. For `closureEnvironment`, indicate what the variable points to.
- You do not need to include the chunk header information (size and number of pointer variables).
- You do not need to include the parameter chunk or frame for the special "start" procedure automatically added by the compiler; assume the first two chunks allocated are the parameter chunk of `main` followed by the frame of `main`.

Solution: Here is the solution based on my implementation of `newPair`. If your solution is different (for example if you didn't use local variables for the pair elements) your diagram might be a bit different. The important parts are:

- `main` is on the stack and has a static link of zero (since it's not nested in anything).
- `newPair` is on the heap, and also has a static link of zero.
- The closure chunk for `selectMethod` is on the heap, after the parameter chunk and frame of `newPair`.
- The closure chunk's environment is the frame of `newPair`.
- `main`'s frame has a pointer to the closure chunk for `selectMethod` stored in `pair`.

Heap starts below

`newPair` parameter chunk
`a`
`b`
static link, points to address zero

`newPair` frame
`first`
`second`
standard frame variables

closure chunk for `selectMethod` `closureCode`, address of `selectMethod` `closureEnvironment`, points to `newPair` frame
⋮ ⋮ ⋮
`main` frame `pair`, points to closure chunk for `selectMethod` standard frame variables

`main` parameter chunk
`a`
`b`
static link, points to address zero

Stack starts above

2 Tail Calls

Consider the following program:

```
def main(a:Int,b:Int):Int = {
    tailCallOne(a,b)
}
def tailCallOne(a:Int,b:Int):Int = {
    tailCallTwo(a,b)
}
def tailCallTwo(a:Int,b:Int):Int = {
    a+b
}
```

Suppose `main(1,2)` is executed.

1. Assuming *no* tail call optimization is performed, which procedure frames or parameter chunks are stored on the stack at the program point right before `tailCallTwo` executes the line `a+b`?

   Solution: The frame and parameter chunk of `main`, the frame and parameter chunk of `tailCallOne`, and the frame and parameter chunk of `tailCallTwo`.

2. Assuming that tail call optimization *is* performed, which procedure frames or parameter chunks are stored on the stack at the program point right before `tailCallTwo` executes the line `a+b`?

3. Consider the following modified program. Now `tailCallOne` is a nested procedure of `main` with no parameters, and it accesses the parameters of `main`.

   ```
   def main(a:Int,b:Int):Int = {
       def tailCallOne():Int = {
           tailCallTwo(a,b)
       }
       tailCallOne()
   }
   def tailCallTwo(a:Int,b:Int):Int = {
       a+b
   }
   ```

   Explain why performing tail call optimization on the call to `tailCallOne` is invalid.

   Solution: The frame and parameter chunk of `main` are stored on the stack. The tail call optimization will cause these chunks to be popped off the stack before calling `tailCallOne`. The frame and parameter chunk of `tailCallOne` then get pushed on the stack, possibly overwriting the old data from `main`. But `tailCallOne` needs to access the variables from `main`, so this creates a problem. For this reason, the tail call optimization should not be performed on calls to nested procedures in general.

4.
   ```
   def factorial(n:Int):Int = { if(n <= 1) { 1 } else { n * factorial(n-1) } }
   def sum(n:Int,a:Int):Int = { if(n != 0) { sum(n-1,n+a) } else { a } }
   ```

   Is the recursive call to `factorial` a tail call? Is the recursive call to `sum` a tail call? Explain.

   Solution: The call to `factorial` is not a tail call. While it appears at the end of the code, this is not enough to be a tail call – the call has to be the very last thing executed before the epilogue of the caller. In the `factorial` procedure, the last thing that happens before the epilogue will be a multiplication operation.

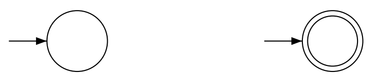
However, the call to `sum` is a tail call. The arguments are evaluated, and then the call is the very last thing that happens before the epilogue of the caller. Note that even though the call to `sum` is not actually at the end of the code due to the `else` clause, it is still a tail call.

In general, you can't tell whether something is a tail call by looking at where it physically appears in the code – you need to think about the semantics and whether anything happens after it is executed.

3 Regular Languages

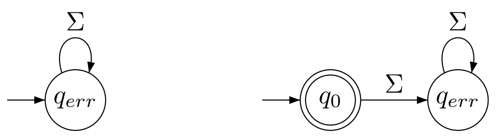
1. Draw DFAs for the empty language $\emptyset$ which contains no words, and the language $\{\varepsilon\}$ that contains only the empty word. Then write the formal definitions for these DFAs (for an arbitrary alphabet Σ).

   Solution: The left DFA recognizes the empty language. The right DFA recognizes $\{\varepsilon\}$.

   Note that to define these DFAs formally, we need to define the transition function for every state-letter pair. The above diagrams do not specify all the transitions. We need to complete the above diagrams by adding the missing transitions and an error state.

   In the case of the DFA for $\emptyset$, we can just use the initial state as the error state, but for $\{\varepsilon\}$ we need a separate error state. We use Σ as a transition label to denote a transition on every letter of the alphabet Σ.

   Formally, the left DFA for $\emptyset$ is $(\Sigma, Q = \{q_{err}\}, q_{err}, A = \emptyset, \delta)$ where $\delta(q_{err}, a) = q_{err}$ for all $a \in \Sigma$.

   Formally, the right DFA for $\{\varepsilon\}$ is $(\Sigma, Q = \{q_0, q_{err}\}, q_0, A = \{q_0\}, \delta)$ where $\delta(q_0, a) = q_{err}$ and $\delta(q_{err}, a) = q_{err}$ for all $a \in \Sigma$.

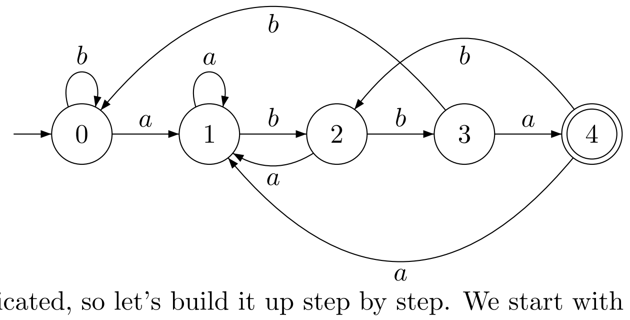
2. Consider the language of all words over $\{a, b\}$ that end with $abba$.

   (a) Write a regular expression for this language.

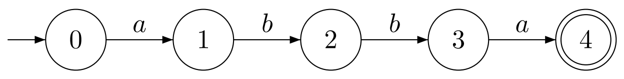
   Solution: $(a|b)^*abba$

   (b) Draw a DFA for this language.

   Solution:

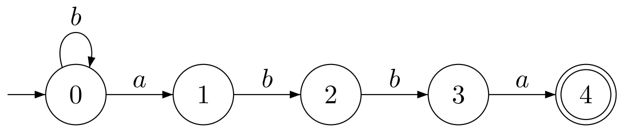
   

   This is a bit complicated, so let's build it up step by step. We start with this DFA as a base.

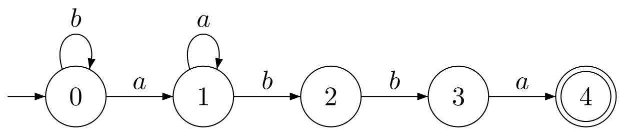


Then we fill in the missing transitions state by state. A key observation is that there's no "error state" in this DFA – no matter what letters we have read so far, it's always possible that we'll eventually read *abba* in sequence.

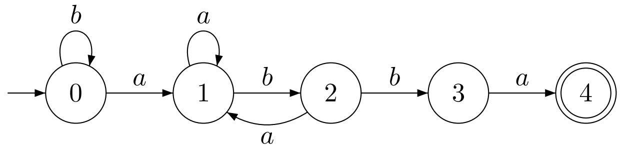
If we are in state 0, you can think of this as meaning we haven't seen any part of *abba* yet. (Technically, what it means is "the longest suffix of the input read so far that is also a prefix of *abba* has length 0", but that's a little harder to wrap your head around.) If we see a *b*, since that's not the first letter of *abba*, we should stay in state 0.



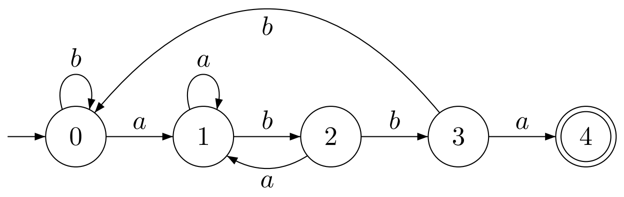
If we're in state 1, we've seen the first *a* of *abba* so far. If we see another *a*, we stay in state 1. Seeing this *a* doesn't bring us forward, but it also doesn't set us back.



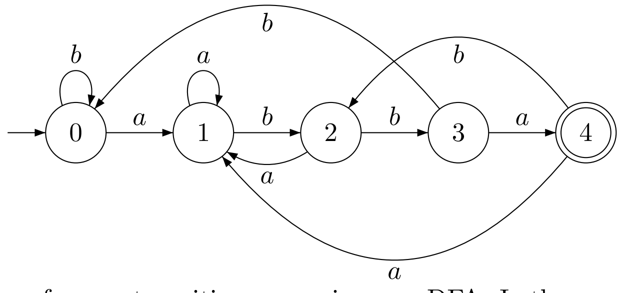
If we're in state 2, we've seen *ab*. If we see *a*, we get *aba* which is not a prefix of *abba*. However, the last *a* could be the start of an *abba*, so we should go back to state 1, which represents having seen an *a*.



If we're in state 3, we've seen *abb*. If we see *b*, we get *abbb*, which can't possibly be the start of *abba*. We have to go back to the beginning, state 0.

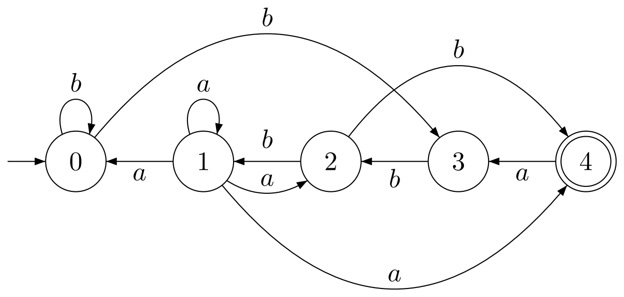


Finally, if we've seen all of *abba*, there are two cases. If we see *a*, we get *abbaa*. The last *a* could be the start of an *abba*, so we go back to state 1. If we see *b*, we get *abbab*. The last *ab* could be the start of an *abba*, so we go back to state 2 which represents having seen *ab*.



(c) Reverse the direction of every transition arrow in your DFA. Is the resulting automaton deterministic or nondeterministic (that is, is it still a DFA or is it now an NFA)?

Solution: Here is the automaton with reversed arrows.

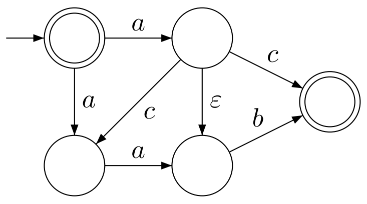


This is nondeterministic. For example, there are four transitions on a out of state 1, and two transitions on b out of states 0 and 2.

3. A NUM token in Lacs consists of a single digit in the range 0–9, or two or more digits the first of which is not 0. Write a regular expression for valid NUM tokens.

Solution: Write $[0-9]$ as shorthand for $(0|1|2|3|4|5|6|7|8|9)$ and similarly $[1-9]$ for $(1|2|3|4|5|6|7|8|9)$. Translating the specification directly gives the expression $[0-9]|[1-9][1-9]^*$. Another valid answer is $0|[1-9][1-9]^*$.

4. This NFA recognizes a finite language. Give a regular expression for the language, then list all the words in the language.



Solution: A regular expression can be constructed by just looking at the different paths through the NFA. One possible answer is $\varepsilon|aab|a(cab|b|c)$. The subexpression ε corresponds to staying in the initial accepting state. The subexpression aab corresponds to taking the downward path from the initial state. The subexpression $a(cab|b|c)$ corresponds to taking the rightward path from the initial state, and then taking one of the three possible paths from that state.

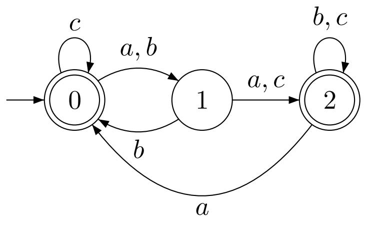
The words in the language are: ε, ab, ac, aab, $acab$.

5. Draw a diagram of the DFA $(\Sigma, Q, q_0, A, \delta)$ where

- $Q = \{0, 1, 2\}$, $\Sigma = \{a, b, c\}$, $q_0 = 0$ and $A = \{0, 2\}$.
- $\delta(q, a) = (q + 1) \bmod 3$

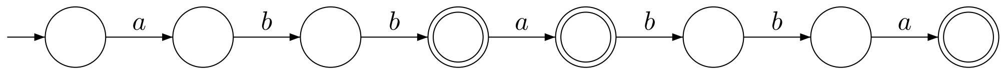
- $\delta(q,b) = \begin{cases} 1-q, & \text{if } q \neq 2; \\ q, & \text{otherwise.} \end{cases}$
- $\delta(q,c) = \begin{cases} 2, & \text{if } q = 1; \\ q, & \text{otherwise.} \end{cases}$

Solution:



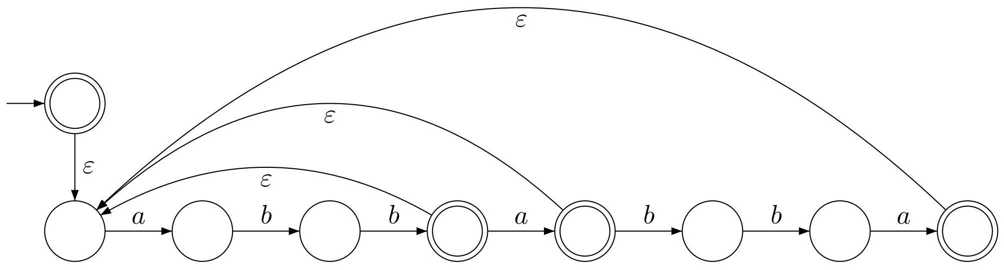
4 Scanning

Here is a DFA.



1. Suppose L is the language accepted by the above DFA. Draw a finite automaton that recognizes L^*. You can (and should) use ε-transitions. (Hint: Don't forget that L^* always contains ε.)

   Solution: The basic idea is to add ε-transitions from each accepting state back to the initial state, as discussed in class. However, there's one extra trick – because L^* contains ε, we must add an extra state which is initial and accepting to accept ε. We then have an ε-transition from this new state to the original initial state.



2. We say a word w can be *scanned with respect to* L if there exist words $w_1, \ldots, w_n \in L$ such that $w = w_1 \cdots w_n$. Explain how your finite automaton for L^* could be used to implement an algorithm that checks if a word can be scanned with respect to L.

   You don't need to give a step-by-step description of the algorithm or pseudocode for the algorithm, just explain how the finite automaton for L^* could help with the task.

   Solution: By the definition of L^*, a word can be scanned with respect to L if and only if it is contained in L^*. Therefore, you can check if a word can be scanned by just using a recognition algorithm for the NFA for L^*.

   Note: If your answer for this question was something to do with using Maximal Munch, your answer is probably wrong. Maximal Munch sometimes rejects words even when it is possible to scan them, so it is not a reliable way to check if a word can be scanned.

3. Give an example of a word over $\{a, b\}$ that cannot be scanned with respect to L.

   Solution: The language L is $\{abb, abba, abbabba\}$, so any word that is not created by concatenating these words together will work. For example, ε, a, b, and ab all cannot be scanned with respect to L.

4. Give an example of a word that can be scanned with respect to L in multiple ways. That is, there should be at least two *different* sequences of words $w_1, \ldots, w_m \in L$ and $w'_1, \ldots, w'_n \in L$ such that $w = w_1 \cdots w_m = w'_1 \cdots w'_n$.

   Solution: *abbabba* is an example. It can be scanned as two tokens *abb*, *abba* or as a single token *abbabba*.

5. Determine the result of using Maximal Munch with this DFA on the following input words. If the scan is successful, give the resulting sequence of tokens. If it is not successful, give the tokens that were scanned before the error occurred, followed by "ERROR". Additionally, if Maximal Munch is not successful, indicate whether the word can be scanned with respect to L, and if it can, describe a valid scan of the word.

   (a) *abbaabbabb*

   Solution: Maximal munch produces: *abba abba* ERROR.
   A valid scan is *abba abb abb*.

   (b) *abbaabbabbabba*

   Solution: Maximal munch produces: *abba abbabba* ERROR.
   A valid scan is *abba abb abb abba*.

   (c) *abbabbaabbaabba*

   Solution: Maximal munch produces: *abbabba abba abba*.

   (d) *abbabbaabbabbabbabba*

   Solution: Maximal munch produces: *abbabba abbabba* ERROR.
   A valid scan is *abbabba abb abb abb abba*.

5 Context-Free Languages

1. Give a context-free grammar for the language of balanced sequences of parentheses (sequences where every left parenthesis has a matching right parenthesis). For example, the sequence (()()(()))() is balanced, but the sequence (()(is not.

   Solution: One possible answer is the following grammar:

   $$S \to \varepsilon \mid SS \mid (S)$$

   The rules $S \to (S)$ and $S \to \varepsilon$ let you generate all strings of the form $(((\cdots)))$, the rule $S \to SS$ lets you put these strings in sequence, and then $S \to (S)$ can be used again to nest sequences. It's clear that every string generated by this grammar is a balanced sequence of parentheses. It's a little harder to see that every balanced sequence of parentheses can be generated this way, and formally proving this is outside the scope of the course, but hopefully the idea is intuitive.

2. Consider the following context-free grammar:

   $$S \to \varepsilon \mid aSbS \mid bSaS$$

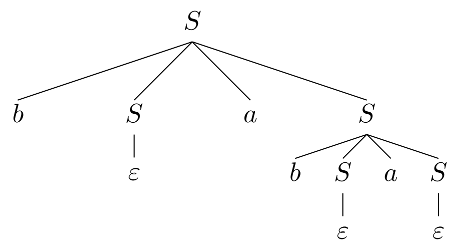
   (a) For each of the following words, give a derivation of the word, and draw a parse tree.

i. *baba*

Solution: A possible derivation is

$$S \Rightarrow bSaS \Rightarrow baS \Rightarrow babSaS \Rightarrow babaS \Rightarrow baba.$$

The corresponding parse tree is below.

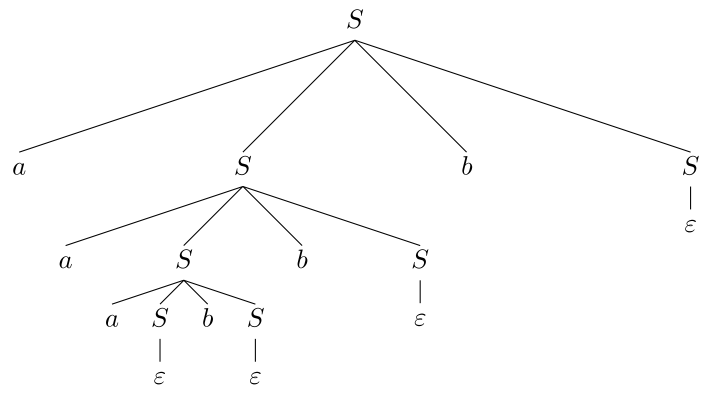


ii. *aaabbb*

Solution: A possible derivation is

$$S \Rightarrow aSbS \Rightarrow aaSbSbS \Rightarrow aaaSbSbSbS \Rightarrow aaabSbSbS \Rightarrow aaabbSbS \Rightarrow aaabbbS \Rightarrow aaabbb.$$

The corresponding parse tree is below.

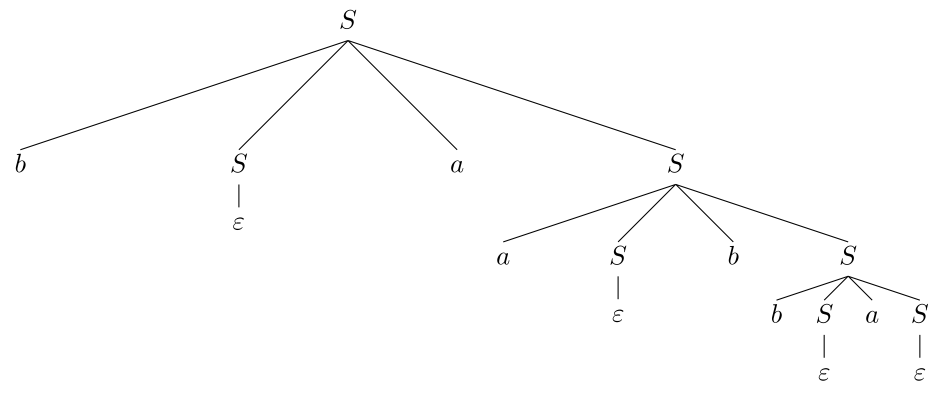


iii. *baabba*

Solution: A possible derivation is

$$S \Rightarrow bSaS \Rightarrow baS \Rightarrow baaSbS \Rightarrow baabS \Rightarrow baabbSaS \Rightarrow baabbaS \Rightarrow baabba.$$

The corresponding parse tree is below.

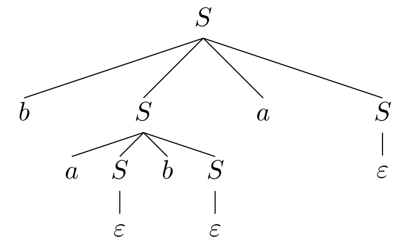


(b) Describe the language this grammar generates.

Solution: It is clear from the structure of the grammar that every word in the language has a equal number of occurrences of a and b. In fact, this grammar generates the language of *all* words over $\{a, b\}$ with an equal number of occurrences of a and b. Formally proving this is outside the scope of this course, but the intuition is similar to the grammar for balanced parentheses. The basic building blocks are words of the form $aaa \cdots bbb$ or $bbb \cdots aaa$. Clearly the grammar generates all such words. We can also use the grammar to form sequences of them, and to nest them inside each other. This is enough to get all words with an equal number of occurrences of a and b.

(c) Prove that this grammar is ambiguous.

Solution: It suffices to find two different parse trees for the same word. Here is another parse tree for the word *baba*.



3. Give an example of a context-free language that is not regular, and an example of a language that is not context-free.

Solution: The language of words of balanced parentheses is an example of a context-free but non-regular language. Another example is arithmetic expressions with parentheses. The language of words with an equal number of occurrences of a and b is yet another example. Generally, any language that allows for “nesting” will not be regular.

An example of a language that is not context-free is the language of all valid Lacs programs. It is not context-free because name and type correctness cannot be checked by a context-free grammar, which is why we have a separate “context-sensitive analysis” phase in our compiler to do these checks.

6 Parsing

1. The following grammar for arithmetic expressions with addition, multiplication and the number two is ambiguous:

$$\begin{aligned} \mathit{expr} &\rightarrow \mathit{expr}\ \mathit{op}\ \mathit{expr} \\ \mathit{expr} &\rightarrow 2 \\ \mathit{op} &\rightarrow + \mid * \end{aligned}$$

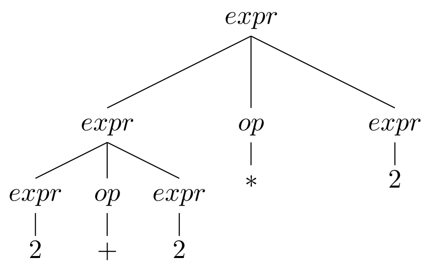
This does not cause a problem for the CYK parsing algorithm. The CYK algorithm can parse ambiguous grammars and will return one of the possible parse trees. However, depending on what you do with the parse tree, the result could be undesirable.

(a) Consider an expression evaluator that works as follows. First, it runs the CYK parsing algorithm on the expression to produce a parse tree. Next, it recursively evaluates the expression using the returned parse tree, as follows:

- If the production at the root of the tree is $expr \rightarrow expr\ op\ expr$, recurse on the two child $expr$ nodes to determine their value, then perform the operation derived from op on the two values and return the result.
- If the production at the root of the tree is $expr \rightarrow 2$, return 2.

Give a parse tree for the expression $2 + 2 * 2$ which will cause this expression evaluator to return an incorrect result.

Solution: Here is such a tree. The structure of this tree means addition will be performed before multiplication, which is wrong.

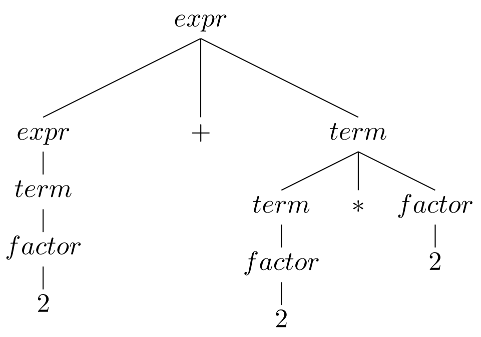


(b) Give a different grammar for this expression language that is guaranteed to produce correct parse trees. You may introduce additional non-terminal symbols and assume the expression evaluator will handle them appropriately. (Hint: look at the Lacs grammar for inspiration.)

Solution: Similarly to the Lacs grammar, we divide expressions into $expr$, $term$ and $factor$. The structure of the grammar makes it so that multiplication always appears deeper in the parse tree than addition.

$$\begin{aligned} expr &\rightarrow expr + term \mid term \\ term &\rightarrow term * factor \mid factor \\ factor &\rightarrow 2 \end{aligned}$$

The expression $2 + 2 * 2$ has a unique parse tree under this grammar.



2. This question is about the inner workings of the CYK parser. Here is a brief review of how the CYK parser works. The CYK parser takes two inputs: a grammar, and a sequence of terminal symbols. It then calls a recursive function which does the actual parsing work. This recursive function, which we will call R, takes two arguments:
   - A sequence of terminals and non-terminals, which we will call α.
   - A substring of the input, which we will call x.

The function R works as follows. If $\alpha \Rightarrow^* x$, then R returns a sequence of parse trees, one for each symbol in α, such that leaves of the trees spell out x when concatenated together. Otherwise, R returns a value to signal that the parse failed (in Scala we use "None" for this).

Suppose we use the following grammar as input to our CYK parser:

$$\begin{aligned} &S \to A \mid SA \\ &A \to a \end{aligned}$$

For each of the following calls to R, draw the sequence of parse trees resulting from the call, or write "None" if the parse fails.

(a) $R(A, a)$

Solution:

$$\begin{tikzcd} A \arrow[d, dash] \\ a \end{tikzcd}$$

Explanation: The parser will try the rule $A \to a$ and then construct a tree of the form

$$\begin{tikzcd} A \arrow[d, dash] \\ R(a, a) \end{tikzcd}$$

The recursive call $R(a, a)$ is of course a successful parse, and just returns a tree consisting of a single node a.

(b) $R(S, a)$

Solution:

$$\begin{tikzcd} S \arrow[d, dash] \\ A \arrow[d, dash] \\ a \end{tikzcd}$$

Explanation: The parser will try the rules $S \to A$ and $S \to SA$ in some order. The application of $S \to SA$ will not result in an successful parse. However, $S \to A$ will work and the following tree will be constructed:

$$\begin{tikzcd} S \arrow[d, dash] \\ R(A, a) \end{tikzcd}$$

(c) $R(aA, aa)$

Solution:

$$\begin{tikzcd} a & A \arrow[d, dash] \\ & a \end{tikzcd}$$

Explanation: Because α starts with a terminal, and this terminal matches the one at the start of x, the parser will return the sequence of trees $Seq(a, R(A, a))$.

(d) $R(SA, aa)$

Solution:

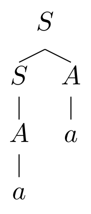
$$\begin{tikzcd} S \arrow[d, dash] & A \arrow[d, dash] \\ A \arrow[d, dash] & a \\ a & \end{tikzcd}$$

Explanation: Because α starts with a nonterminal, but has length greater then one, the parser will look at all the different ways of splitting the string x into two parts:

- Split it into ε and aa, do recursive calls $R(S, \varepsilon)$ and $R(A, aa)$. This fails.
- Split it into a and a, do recursive calls $R(S, a)$ and $R(A, a)$. This succeeds and the above sequence of two trees is returned.
- If the previous attempt hadn't succeeded, the parser would next look at the split aa and ε, and try $R(S, aa)$ and $R(A, \varepsilon)$.

(e) $R(S, aa)$

Solution:



Explanation: The parser will try the rules $S \to A$ and $S \to SA$ in some order. The application of $S \to A$ will not result in an successful parse. However, $S \to SA$ will work, and a tree will be constructed where the root is S and the children come from the sequence returned by $R(SA, aa)$.

3. Give an example of an ambiguous grammar such that there are infinitely many parse trees for a single word. Explain how the CYK parsing algorithm is able to parse this grammar in finite time.

An example of a grammar with infinitely many parse trees for a single word is:

$$S \to A$$
$$A \to S$$
$$A \to a$$

Your parse tree can cycle between S and A an arbitrary number of times before finally deriving a.

The CYK parser handles this using memoization: it remembers which inputs it has attempted to parse before to avoid getting into cycles. For example, here is what will happen when the recursive parsing function R is called with inputs $\alpha = S$ and $x = a$:

- $R(S, a)$ is called. The CYK parser makes note of the fact that it has seen the pair (S, a) in its memoization table by storing the value "None" as the result of the parse. It will later update this "None" to the real value if the parse $R(S, a)$ is actually successful, but for now it just marks the input as seen.
- The parser tries the rule $S \to A$ and calls $R(A, a)$.
- The parser tries the rule $A \to S$. But it realizes it has seen the input (S, a) before, and returns the value "None" from the memoization table instead of actually calling $R(S, a)$. So this attempt at entering a cycle is treated as a parsing failure.
- The parser then tries the rule $A \to a$ and calls $R(a, a)$, which leads to a successful parse.

4. Give an example of a grammar that cannot be parsed by an LR parser.

Solution: LR parsers cannot parse ambiguous grammars, so any ambiguous grammar will work.

5. Imagine you are designing the Marmoset tests for the Earley parsing bonus problem. You want to make sure that students cannot pass the tests by just making the `parseEarley` function do a call to `parseCYK`. What differences between the CYK and Earley algorithms could you exploit to achieve this?

Solution: The Earley parser is significantly faster than the CYK parser for many grammars. While the worst case running time is cubic for both parsers, the Earley parser is able to achieve quadratic or linear time on some grammars that CYK handles more slowly. You could design a large test case that is highly likely to time out when parsed with CYK, but can be parsed in linear time with Earley.

6. Define the correct prefix property and explain why it is useful for a programming language parser to have this property.

   Solution: Informally, a parser has the correct prefix property if it rejects the input as soon as it possibly can: the moment the parser finds a prefix of the input which cannot be extended to a valid input, it rejects the input.

   Formally, suppose L is the language to be parsed. Suppose the following statements are true:

   - The word $w = xaz$ is not in L, where $w, x, z \in \Sigma^*$ and $a \in \Sigma$.
   - The word x is a prefix of a word in L, that is, there exists $y \in \Sigma^*$ such that $xy \in L$.
   - The word xa is not a prefix of a word in L, that is, there does not exist $v \in \Sigma^*$ such that $xav \in L$.

   A parser has the *correct prefix property* if for all $w = xaz$, the word xaz is rejected as soon as the parser processes xa.

   This is a useful property for a programming language parser because it enables better error reporting: the parser can tell the user the exact location at which the program "stops being valid", which is often close to where the syntax error actually appears.

7 Context-Sensitive Analysis

1.
```
def main(a:Int,b:Int):Int = { f() }
def f():Int = {
    def f():Int = { 0 }
    f()
}
```

   Will this Lacs code compile? If it compiles and runs, will it recurse infinitely, or return zero? Explain.

   Solution: This code will compile, and it will return zero.

   To understand why, you need to understand how the symbol tables of procedures are constructed. Let's first look at this similar program, where the inner occurrence of `f` has been renamed to `g`.

```
def main(a:Int,b:Int):Int = { f() }
def f():Int = {
    def g():Int = { 0 }
    g()
}
```

   The top-level symbol table will contain `main` and `f`. The symbol table of `f` will contain `g`, because `g` is nested inside `f`. The symbol table of `f` also inherits all the names from the top-level symbol table, so it has access to `main` and `f`.

   The same principles apply to the original program, with one extra wrinkle. The top-level symbol table will contain `main` and the "outer" `f`. The symbol table of "outer" `f` will contain "inner" `f`, because "inner" `f` is nested inside "outer" `f`. The symbol table of "outer" `f` also inherits all the names from the top-level symbol table. But here is the difference: the symbol table of "outer" `f` will not contain "outer" `f`, because this name has been overriden by "inner" `f`.

So when "outer" `f` calls the procedure named `f`, it looks in its symbol table, and finds "inner" `f`, resulting in the call returning 0.

2. In this question, you are asked to write out the symbol tables of some Lacs procedures. The symbol table should contain all the variable and procedure names defined in the procedure (including parameters), and it should specify whether each name corresponds to a procedure or a variable.

   Include type information about each variable and procedure. For example, for the procedure `f` below you would state that it is of type `(Int,Int)=>Int`.

```
def f(a:Int, b:Int):Int = {
    var c:Int;
    def g(a:Int, b:(Int)=>Int):Int = {
        b(a)
    }
    def h(c:Int):Int = {
        def g():Int = {
            c-b
        }
        g()
    }
    c = a+b;
    g(c,h)
}
```

   (a) Write out the symbol table of `f`.

   Solution:

Name	Information
`f`	procedure of type `(Int,Int)=>Int`
`g`	procedure of type `(Int,(Int)=>Int)=>Int`
`h`	procedure of type `(Int)=>Int`
`a`	variable of type `Int`
`b`	variable of type `Int`
`c`	variable of type `Int`

   (b) Write out the symbol table of `g`. Indicate which variables are inherited from the symbol table of `f` and which ones were overriden by `g`.

   Solution:

Name	Information	Inherited/Overriden
`f`	procedure of type `(Int,Int)=>Int`	inherited from `f`
`g`	procedure of type `(Int,(Int)=>Int)=>Int`	inherited from `f`
`h`	procedure of type `(Int)=>Int`	inherited from `f`
`a`	variable of type `Int`	overriden by `g`
`b`	variable of type `(Int)=>Int`	overriden by `g`
`c`	variable of type `Int`	inherited from `f`

   (c) Write out the symbol table of `h`. Indicate which variables are inherited from the symbol table of `f` and which ones were overriden by `h`.

   Solution:

Name	Information	Inherited/Overriden
`f`	procedure of type `(Int,Int)=>Int`	inherited from `f`
`g`	procedure of type `()=>Int`	overriden by `h`
`h`	procedure of type `(Int)=>Int`	inherited from `f`
`a`	variable of type `Int`	inherited from `f`
`b`	variable of type `Int`	inherited from `f`
`c`	variable of type `Int`	overriden by `h`

3. Check the body of each of the following procedures for type correctness. If the body has no type errors, state the return type of the procedure. Otherwise, describe the type error.

(a)
```
def f(b:(Int,Int)=>Int):??? = {
    b(b(1,2),b(3,4))
}
```

Solution: Correct. The return type is `Int`.

(b)
```
def f(b:(Int,Int)=>Int):??? = {
    b(1,b(2,3),4)
}
```

Solution: Type error. The procedure `b` takes 2 arguments, but it is called with 3.

(c)
```
def f(a:Int,b:(Int)=>(Int)=>Int):??? = {
    if(a>0) {
        b(a)
    } else {
        b(0)
    }
}
```

Solution: Correct. The return type is `(Int)=>Int`.

(d)
```
def f(a:Int,b:(Int)=>(Int)=>Int):??? = {
    if(a>0) {
        b(a)
    } else {
        b
    }
}
```

Solution: Type error. The "if clause" and "else clause" of the if statement have different types, which is not allowed.

(e)
```
def f(a:(Int)=>(Int,Int)=>Int,b:(Int)=>(Int,Int)=>Int,c:Int):??? = {
    var d:((Int,Int)=>Int,Int)=>(Int)=>Int;
    def e(a:(Int,Int)=>Int,b:Int):(Int)=>Int = {
        a(b,c)
    }
    d = e;
    e(if(c>0) { a(c) } else { b(c) }, c)
}
```

Solution: Type error. The call to `a` within the procedure `e` returns an `Int`. This does not match the return type of `e`, which is `(Int)=>Int`.

(f)
```
def f(a:(Int)=>(Int,Int)=>Int,b:(Int)=>(Int,Int)=>Int,c:Int):??? = {
    var d:((Int,Int)=>Int,Int)=>(Int)=>Int;
    def e(a:(Int,Int)=>Int,b:Int):(Int)=>Int = {
        def d(c:Int):Int = {
            a(b,c)
        }
        d
    }
    d = e;
    e(if(c>0) { a(c) } else { b(c) }, c)
}
```

Solution: Correct. The return type is the return type of `e`, which is `(Int)=>Int`.

8 Memory Management

1. In this question, we will use the "explicit allocation and deallocation" method of memory management. In this method two functions are provided:

   - `malloc(requested_bytes)`, which allocates a block of size `requested_bytes + 8` (8 bytes for a header, and the rest free to use) and returns the address of the block. If there is not enough space for the block, or `requested_bytes` is not a positive multiple of 4, an error is returned.
   - `free(block_address)`, which frees the block at the specified address. If there are free blocks adjacent to the block that was just freed, these blocks will be coalesced (merged) into a single free block.

   Suppose we have an empty heap with 64 bytes of available memory (not including the "dummy block" at the start, which consists only of an 8-byte header and is used to make it easier to implement `malloc` and `free`).

   Describe a sequence of `malloc` and `free` calls that results in the heap having 24 or more bytes of free space, but enough fragmentation that a `malloc(8)` call will fail.

   Solution: Here is one possible solution:

   ```
   a = malloc(4)
   b = malloc(8)
   c = malloc(4)
   d = malloc(16)
   free(a)
   free(c)
   ```

   In the resulting heap, the only free blocks have size 12 (4 plus the 8 byte header) and they are surrounded by used blocks or by non-heap space. This means `malloc`(8), which requires a free block of size 16, will fail.

2. Below is a diagram of a small stack and heap. The heap contains 128 bytes in total and is divided into two 64 byte semispaces. The heap starts at address 128. Currently, the first semispace (lower addresses) is the "from-space" and the second semispace (higher addresses) is the "to-space".

   Suppose garbage collection is performed with the stack and heap in this state. This will result in all the reachable blocks in the first semispace of the heap being copied into the second semispace. Additionally, the size of each reachable block in the first semispace will be negated (to mark the block as copied), and the "number of pointers" field in each reachable block will be overwritten with the block's new address. All reachable pointers to the moved block should be updated. Unreachable blocks in the first semispace are not modified.

Fill out the "After Garbage Collection" diagram of the heap and stack with the correct values.

The order in which the chunks are copied to the heap matters to an extent. You should first copy the chunks that are reachable from the stack, and then the chunks that are reachable from those chunks, and so on. However, it does not matter whether (for example) you scan the stack from top to bottom or from bottom to top, as this is just an implementation detail that does not affect the correctness of the algorithm.

Stack
32
4
128
152
128
192
140
42

Before Garbage Collection			
Heap Semispace 1		Heap Semispace 2	
Address	Value	Address	Value
128	12	192	0
132	1	196	0
136	168	200	0
140	12	204	0
144	1	208	0
148	140	212	0
152	16	216	0
156	2	220	0
160	128	224	0
164	168	228	0
168	24	232	0
172	2	236	0
176	152	240	0
180	0	244	0
184	128	248	0
188	140	252	0

After Garbage Collection			
Heap Semispace 1		Heap Semispace 2	
Address	Value	Address	Value
128		192	
132		196	
136		200	
140		204	
144		208	
148		212	
152		216	
156		220	
160		224	
164		228	
168		232	
172		236	
176		240	
180		244	
184		248	
188		252	

Stack

Solution:

- Start by scanning the stack. We'll go from top to bottom. The first pointer (of 4) is to 128. We copy the corresponding 3-word chunk to the start of Semispace 2. We negate the size and put in the "forwarding address" to 192. We update the 128 on the stack to 192.
- Next is 152. Copy the 4-word chunk and update everything as before.
- Next is 128 again. We've already copied this chunk, so we look at the forwarding address, which is 192, and just update the pointer on the stack.
- Next is 192. This address is not in the from-space, so we leave it alone.
- That's all the pointers on the stack. In particular, you shouldn't have copied the chunk at 140 because 140 is the value of a non-pointer variable.
- Now scan the to-space, top to bottom again. First chunk has one pointer, to 168. Copy the chunk and update everything.
- Next chunk has two pointers. First is 128. We copied this chunk already, so update the pointer to the forwarding address of 192. Next is 168, which was also already copied. Update the pointer to 220.
- Next chunk has two pointers. 152 is a chunk we've copied, so we update it to 204. The other pointer is 0, which is not in the from-space, so we ignore it. Ignore the non-pointers 128 and 140.

Now we're done because we've scanned all the chunks in the to-space. The result is below.

After Garbage Collection			
Heap Semispace 1		Heap Semispace 2	
Address	Value	Address	Value
128	-12	192	12
132	192	196	1
136	168	200	220
140	12	204	16
144	1	208	2
148	140	212	192
152	-16	216	220
156	204	220	24
160	128	224	2
164	168	228	204
168	-24	232	0
172	220	236	128
176	152	240	140
180	0	244	0
184	128	248	0
188	140	252	0

Stack
32
4
192
204
192
192
140
42

3. (a) Define what it means for a block in memory to be live.

   Solution: A block is live if it will be accessed at some later point in the program.

   (b) Define what it means for a block in memory to be reachable.

   Solution: A block is reachable if the address is stored in the stack or in a register, or (recursively) the address is stored in a reachable block.

   (c) If a block is reachable, is it necessarily live?

   Solution: No. It is possible that the address of a block is stored somewhere, but that address will never be used for the rest of the program.

   (d) If a block is live, is it necessarily reachable?

   Solution: Yes. If a block will be accessed at some point in the future, the address of that block must be stored somewhere, or else it won't be possible to access it.

   (This is assuming your program is operating at a high level, like Lacs programs. If you are writing low level code that can directly access memory, then essentially everything is "reachable" because you can read and write from arbitrary addresses.)

   (e) If a block is unreachable, can it be live?

   Solution: No. An unreachable block can never be accessed again because there are no more references to its address anywhere, so the block cannot be live.

   (f) If a block is not live, can it be reachable?

   Solution: Yes. It is possible that a block will never be used again, but there is still a reference to its address somewhere.