CS241E Final Exam Review

Sylvie Davies

University of Waterloo

December 17, 2019

These Slides

If you want them, these slides can be downloaded from:

https://www.student.cs.uwaterloo.ca/~cs241e/current/FinalReviewSlides.pdf

Closures

- A closure consists of two pieces of information:
  - A procedure. which may contain some free variables.
    Free variables are variables that are declared outside of the procedure.
  - An environment, which assigns values to all of the free variables, transforming them to bound variables.
- Example:

```
def procedure(a:Int,b:Int):Int = {
    var d:Int;
    d = 4;
    a + b + c + d + e
}
```

  The variables c and e are free variables. The procedure cannot be called until these variables are bound.

- This is the abstract definition of a closure. We will talk about implementation shortly, but for now, just think of a closure as a procedure and a binding of free variables to values.

Closures

- A closure consists of two pieces of information:
  - A procedure. which may contain some free variables.
    Free variables are variables that are declared outside of the procedure.
  - An environment, which assigns values to all of the free variables, transforming them to bound variables.
- It is possible for the environment to be modified after it is created:

```
def counter():Int = {
    count = count + 1;
    count
}
```

Here `count` is a free variable. Suppose when the closure is created, `count` is bound to 0. Each call will modify the value of `count` stored in the environment.

```
// assume count is bound to 0
var newCounter:()=>Int;
newCounter = counter;
newCounter(); // returns 1
newCounter(); // returns 2
```

Closure Creation

- To create a closure, we need to allocate a chunk in memory containing both the procedure and the environment.
- Instead of storing the entire procedure, we simply store the address of the procedure's code.
- Since a closure can access any variable declared in any outer procedure, the environment we create must contain data from all active frames and parameter chunks of every outer procedure of the closure procedure at closure creation time.
- Instead of storing all this data, we simply store the static link of the closure procedure, which points to the frame of the directly enclosing procedure. We can access other frames by following static links and access parameter chunks by following the parameter pointers.
- So a closure consists of a two-variable chunk containing the address of the procedure to call and the address of the relevant frame of the directly enclosing procedure.

Calling a Closure

- The implementation of a closure consists of a two-variable chunk containing:
  - The address of the procedure to call.
  - The environment, represented as the address of a frame of the directly enclosing procedure.
- When calling a closure, we are given one of these chunks. We simply need to unpack the data.
- We know which procedure to jump to because its address is stored in the chunk.
- To give the procedure access to its environment, we retrieve the environment address from the chunk and pass it in as the static link.
- The closure procedure now has access to everything in the environment via the usual method of following static links to find variables in outer procedures.
- Actually implementing this is tricky (need to make sure not to overwrite important registers!)

Stack or Heap

- Consider the following program:

```
def main(a:Int,b:Int):Int = {
    var myCounter:()=>Int;
    myCounter = newCounter(0); // stores a closure chunk in myCounter
    myCounter(); // returns 1
    myCounter(); // returns 2
}
def newCounter(startingValue:Int):()=>Int = {
    var count:Int;
    def counter():Int = { count = count + 1; count }
    count = startingValue; counter
}
```

- The frame and parameter chunk of `main` can be stored on the stack, but the frames and parameter chunks of `newCounter` and `counter` must be stored on the heap.
- When `newCounter(0)` is called, it returns a closure chunk containing the address of `counter` and the address of the frame of the outer procedure enclosing counter (the frame of `newCounter` itself).

Stack or Heap

- Consider the following program:

```
def main(a:Int,b:Int):Int = {
    var myCounter:()=>Int;
    myCounter = newCounter(0); // stores a closure chunk in myCounter
    myCounter(); // returns 1
    myCounter(); // returns 2
}
def newCounter(startingValue:Int):()=>Int = {
    var count:Int;
    def counter():Int = { count = count + 1; count }
    count = startingValue; counter
}
```

- When `newCounter(0)` returns, its frame needs to be preserved so that closure calls to `counter` have access to their environment.
- If the frame of `newCounter(0)` was on the stack, it would be popped off as soon as `newCounter(0)` returns.
- In general, if a procedure is made into a closure, then the frames and parameter chunks of all of its outer procedures must be allocated on the heap (they are part of the environment).

Stack or Heap

- Consider the following program:

```
def main(a:Int,b:Int):Int = {
    var myCounter:()=>Int;
    myCounter = newCounter(0); // stores a closure chunk in myCounter
    myCounter(); // returns 1
    myCounter(); // returns 2
}
def newCounter(startingValue:Int):()=>Int = {
    var count:Int;
    def counter():Int = { count = count + 1; count }
    count = startingValue; counter
}
```

- Why does the frame of `counter` need to be on the heap?
- It technically doesn't, but when we do the call `myCounter()`, we don't know what procedure we are actually calling. We just have the address of its code (some number).
- The procedure we call via a closure could require heap allocation, or it could not, and we don't know at runtime. To be safe, we always put the frame and parameter chunk on the heap when doing a closure call.

Stack or Heap

- Consider the following program:

```
def main(a:Int,b:Int):Int = {
    var myCounter:()=>Int;
    myCounter = newCounter(0); // stores a closure chunk in myCounter
    myCounter(); // returns 1
    myCounter(); // returns 2
}
def newCounter(startingValue:Int):()=>Int = {
    var count:Int;
    def counter():Int = { count = count + 1; count }
    count = startingValue; counter
}
```

- Why does the frame of `main` go on the stack, even though `main` creates and calls a closure?
- It is a common misconception that procedures which merely create or call closures necessarily need to go on the heap.
- The only procedures which need to go on the heap are procedures which are made into closures, and (recursively) outer procedures of procedures that go on the heap. This does not apply to `main`.

Memory Diagram

After the line `myCounter = newCounter(0)`:

Heap starts below
`newCounter` parameter chunk `startingValue = 0` static link, pointer to address zero
`newCounter` frame `count = 0` standard frame variables
closure chunk for `counter` `closureCode`, address of `counter` `closureEnvironment`, pointer to `newCounter` frame
⋮
`main` frame `myCounter`, pointer to closure chunk for `counter` standard frame variables

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

Stack starts above

Tail Calls

- A procedure call is a tail call if it is the last thing executed in the procedure before the epilogue.
- This does not necessarily correspond to the physical position of the call within the source code.

```
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 } }
```

- The call to `factorial` is not a tail call. After evaluating the call, a multiplication operation must be performed before the epilogue.
- The call to `sum` is a tail call. Even though it is not at the physical end of the code, it is the last thing that happens before the epilogue in that particular branch of the if statement.
- High level idea of tail call optimization: If `f` calls g, and the call is a tail call, the frame and parameter chunk of `f` can be popped off the stack before calling g.
- Without tail call optimization: `push f`, `push g`, `pop g`, `pop f`.
  With tail call optimization: `push f`, `pop f`, `push g`, `pop g`.

Tail Calls, Nested Procedures, and the Heap

- If the tail call is to a nested procedure, the tail call optimization is unsafe!
- With tail call optimization: `push f`, `pop f`, `push g`, `pop g`.
- If g is nested in `f`, then g may want to access the variables of `f`, so we can't pop `f`.
- There's an exception to this rule: if the tail call is a closure call, the tail call optimization is always safe.
- If g is nested in `f`, and we do a closure call to g, then both `f` and g are on the heap! No worries about losing access to the variables of `f`.
- Tail call optimization has other cases depending on which procedures have their frames and parameter chunks stored on the heap.
  `f` on heap: `allocate f on heap`, `push g`, `pop g`.
  g on heap: `push f`, `pop f`, `allocate g on heap`.
  `f` and g on heap: `allocate f on heap`, `allocate g on heap`.

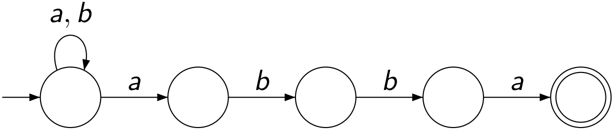
Formal Languages

- An alphabet is a finite non-empty set whose elements are called symbols or letters. Usually denoted Σ.
- A word over Σ is a finite-length sequence of elements of Σ.
- The empty word (sequence of length 0) is denoted ε.
- The set of all words over Σ is written Σ^*.
- A language over Σ is a subset of Σ^*.
- Languages can be finite or infinite.
- We focus on two classes of languages in this course:
  - Regular languages, which can be described by finite automata or regular expressions and are used for scanning in our compiler.
  - Context-free languages, which can be described by context-free grammars and are used for parsing in our compiler.
- We have the following relationship between classes of languages:

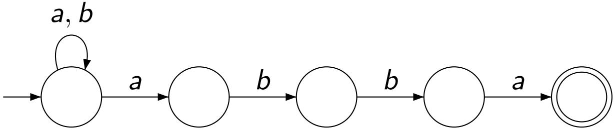
$$\text{finite languages} \subsetneq \text{regular languages} \subsetneq \text{context-free languages}$$

Finite Automata: Informal Definition

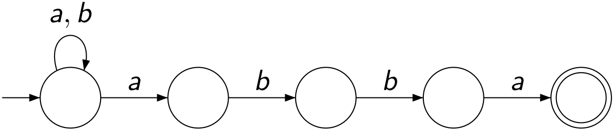
- A finite automaton is collection of "states" that are joined by "transitions". The transitions are generally labelled with letters of the alphabet, but can also be labelled with the empty word ε to signify a transition that is "free to take".
- States can be doubled-circled to mark them as "accepting", and one state is marked as "initial" via an arrow pointing inward.
- Example:



- A word is accepted by the automaton if and only if there is a path from the initial state to an accepting state such that the transition letters spell out the word.
- The language of accepted words here is the set of all words over $\{a, b\}$ that end with *abba*.

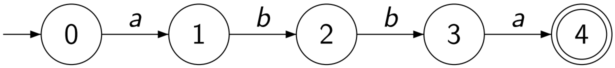


- A finite automaton is deterministic (DFA) if every state has at most one outwards transition on each letter, and there are no ε-transitions.
- This finite automaton is nondeterministic (NFA) because there are two transitions out of the initial state labelled with a.
- Every NFA can be converted to a DFA which accepts the same language, but the DFA may require exponentially more states in the worst case.
- We do not learn the NFA to DFA conversion algorithm in this course, but you may be asked to convert simple NFAs to DFAs just from first principles.



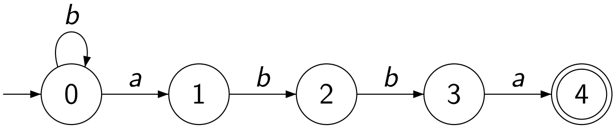
- This NFA recognizes the language of words over $\{a, b\}$ that end with *abba*. Can we find a DFA for the same language?
- Instead of trying to "convert" the NFA, let's just construct a DFA from scratch.

Constructing a DFA



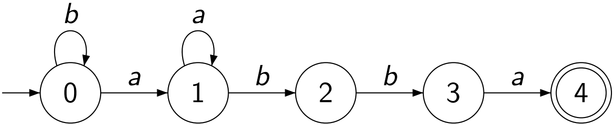
- We start with this DFA as a base.
- This DFA just recognizes the set $\{abba\}$.
- We now fill in the missing transitions to account for other words.

Constructing a DFA



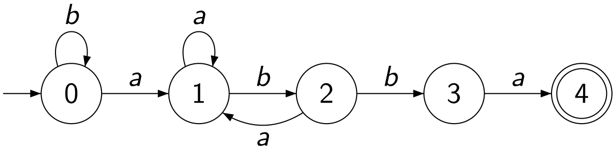
- If we are in state 0, you can think of this as meaning we haven't seen any prefix of *abba* yet.
- If we see a *b*, since that's not the first letter of *abba*, we should stay in state 0.

Constructing a DFA



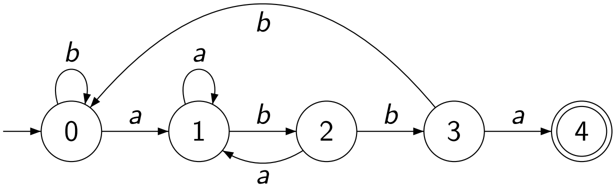
- If we're in state 1, we've seen the first *a* of *abba* so far.
- If we see another *a*, we stay in state 1. Seeing this *a* doesn't bring us forward, but it also doesn't set us back.

Constructing a DFA



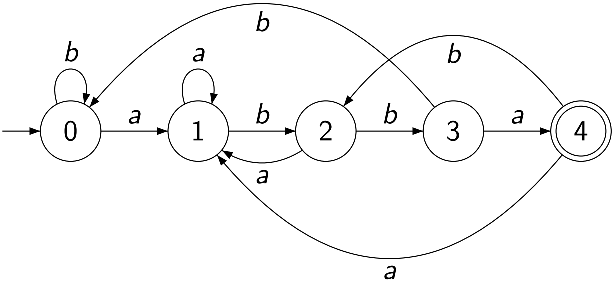
- If we're in state 2, we've seen *ab*.
- If we see *a*, we get *aba* which is not a prefix of *abba*. However, the last *a* could be the start of an *abba*, so we should go back to state 1, which represents having seen an *a*.

Constructing a DFA



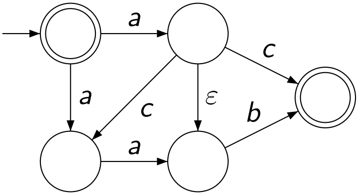
- If we're in state 3, we've seen *abb*.
- If we see *b*, we get *abbb*, which can't possibly be the start of *abba*. We have to go back to the beginning, state 0.

Constructing a DFA



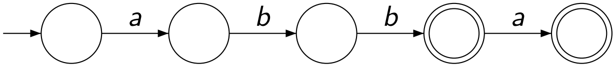
- Finally, if we've seen all of *abba*, there are two cases.
- If we see *a*, we get *abbaa*. The last *a* could be the start of an *abba*, so we go back to state 1.
- If we see *b*, we get *abbab*. The last *ab* could be the start of an *abba*, so we go back to state 2 which represents having seen *ab*.
- This is the finished DFA for the language of words over $\{a, b\}$ that end with *abba*.

ε-Transitions

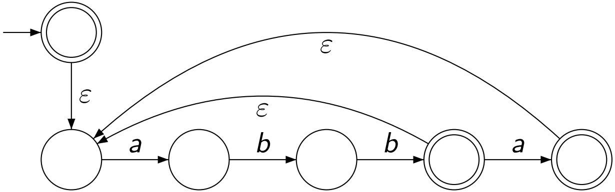


- This NFA recognizes the following words:
- ε
- ab
- ac
- aab
- $acab$

More ε-Transitions



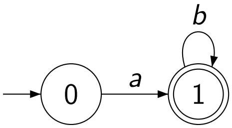
- The above DFA recognizes the language $L = \{abb, abba\}$.



- The above NFA recognizes the language L^* of all words that can be formed by concatenating words in L together, zero or more times.

Finite Automata: Formal Definition

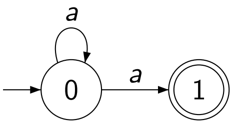
- A finite automaton is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$, where:
  - Σ is an alphabet.
  - Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, and $A \subseteq Q$ is the set of accepting states.
  - δ is the transition function, which is defined differently depending on whether the automaton is a DFA or an NFA:
    DFA: $\delta \colon Q \times \Sigma \to Q$ (given a state and symbol, produce a new state)
    NFA: $\delta \colon Q \times \Sigma \to 2^Q$ (where 2^Q is the set of all subsets of Q)
- When drawing an informal diagram of a DFA, you do not need to draw every transition, but when formally defining a DFA you must define the transition function for every state-symbol pair. Sometimes this requires adding an extra "error" state in the formal definition.



- $\Sigma = \{a, b\}$, $Q = \{0, 1, X\}$, $q_0 = 0$, $A = \{1\}$.
- $\delta(0, a) = 1$, $\delta(1, a) = X$, $\delta(X, a) = X$
- $\delta(0, b) = X$, $\delta(1, b) = 1$, $\delta(X, b) = X$

Finite Automata: Formal Definition

- A finite automaton is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$, where:
  - Σ is an alphabet.
  - Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, and $A \subseteq Q$ is the set of accepting states.
  - δ is the transition function, which is defined differently depending on whether the automaton is a DFA or an NFA:
    DFA: $\delta \colon Q \times \Sigma \to Q$ (given a state and symbol, produce a new state)
    NFA: $\delta \colon Q \times \Sigma \to 2^Q$ (where 2^Q is the set of all subsets of Q)
- For an NFA, missing transitions are specified by using $\emptyset$, the empty set of states, as the result of the transition function.



- $\Sigma = \{a\}$, $Q = \{0, 1\}$, $q_0 = 0$, $A = \{1\}$.
- $\delta(0, a) = \{0, 1\}$, $\delta(1, a) = \emptyset$.

- Don't worry about how NFAs with ε-transitions are formally defined (this was not covered in class).

Finite Automata: Formal Definition of Recognition

- A finite automaton is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$, where:
  - Σ is an alphabet.
  - Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, and $A \subseteq Q$ is the set of accepting states.
  - δ is the transition function, which is defined differently depending on whether the automaton is a DFA or an NFA:
    DFA: $\delta \colon Q \times \Sigma \to Q$ (given a state and symbol, produce a new state)
    NFA: $\delta \colon Q \times \Sigma \to 2^Q$ (where 2^Q is the set of all subsets of Q)
- To formally define the language accepted by a DFA, we extend the transition function to words.
- Define $\delta^* \colon Q \times \Sigma^* \to Q$ as follows:
  - $\delta^*(q, \varepsilon) = q$.
  - If w is a non-empty word, a is the first letter of w, and x is the rest of w, then $\delta^*(q, w) = \delta^*(\delta(q, a), x)$.
- The language recognized by DFA $(\Sigma, Q, q_0, A, \delta)$ is the set $\{w \in \Sigma^* : \delta^*(q_0, w) \in A\}$: all words where the corresponding path goes from the initial state to an accepting state.

Finite Automata: Formal Definition of Recognition

- A finite automaton is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$, where:
  - Σ is an alphabet.
  - Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, and $A \subseteq Q$ is the set of accepting states.
  - δ is the transition function, which is defined differently depending on whether the automaton is a DFA or an NFA:
    DFA: $\delta \colon Q \times \Sigma \to Q$ (given a state and symbol, produce a new state)
    NFA: $\delta \colon Q \times \Sigma \to 2^Q$ (where 2^Q is the set of all subsets of Q)
- The definition for NFAs is similar.
- Define $\delta^* \colon Q \times \Sigma^* \to 2^Q$ as follows:
  - $\delta^*(q, \varepsilon) = \{q\}$.
  - If w is a non-empty word, a is the first letter of w, and x is the rest of w, then $\delta^*(q, w) = \bigcup_{q' \in \delta(q,a)} \delta^*(q', x)$.
- The language recognized by NFA $(\Sigma, Q, q_0, A, \delta)$ is the set $\{w \in \Sigma^* : \delta^*(q_0, w) \cap A \neq \emptyset\}$: all words where at least one corresponding path reaches an accepting state.

Regular Expressions

- In the table below, let $a \in \Sigma$ denote an alphabet symbol, and let R, R' and S denote regular expressions over Σ. The notation $\mathcal{L}(R)$ means the language specified by the regular expression R.

Regular Expression	Corresponding Language
$R = \emptyset$	$\mathcal{L}(R) = \emptyset$
$R = \varepsilon$	$\mathcal{L}(R) = \{\varepsilon\}$
$R = a$	$\mathcal{L}(R) = \{a\}$
$S = R \mid R'$ (Union)	$\mathcal{L}(S) = \mathcal{L}(R) \cup \mathcal{L}(R')$
$S = RR'$ (Concatenation)	$\mathcal{L}(S) = \mathcal{L}(R)\mathcal{L}(R')$
$S = R^*$ (Star)	$\mathcal{L}(S) = \mathcal{L}(R)^*$

- $LL' = \{w \in \Sigma^* : w = xy, x \in L, y \in L'\}$.
- $\{a, b\}\{cc, dd\} = \{acc, add, bcc, bdd\} = \mathcal{L}((a|b)(cc|dd))$.
- If $L = \{w_1, \ldots, w_n\}$ is finite, then $L = \mathcal{L}(w_1| \cdots |w_n)$.
- Example: $\{abc, cab, baca\} = \mathcal{L}(abc|cab|baca)$.

Regular Expressions

- In the table below, let $a \in \Sigma$ denote an alphabet symbol, and let R, R' and S denote regular expressions over Σ. The notation $\mathcal{L}(R)$ means the language specified by the regular expression R.

Regular Expression	Corresponding Language
$R = \emptyset$	$\mathcal{L}(R) = \emptyset$
$R = \varepsilon$	$\mathcal{L}(R) = \{\varepsilon\}$
$R = a$	$\mathcal{L}(R) = \{a\}$
$S = R \mid R'$ (Union)	$\mathcal{L}(S) = \mathcal{L}(R) \cup \mathcal{L}(R')$
$S = RR'$ (Concatenation)	$\mathcal{L}(S) = \mathcal{L}(R)\mathcal{L}(R')$
$S = R^*$ (Star)	$\mathcal{L}(S) = \mathcal{L}(R)^*$

- $L^* = \{w \in \Sigma^* : \exists n \geq 0, w = w_1 \cdots w_n, w_1, \ldots, w_n \in L\}$.
- Equivalently, $L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup \cdots$.
- $(a|b)^*abba$ is the language of words over $\{a, b\}$ that end with $abba$.
- $a(aa)^*$ is the language of words over $\{a\}$ with an odd number of occurrences of a.

Scanning

- Let L be a language. We call the words in this language tokens.
- Scanning is the problem of trying to split a given word into tokens from a given language.
- Given w, find words $w_1, \ldots, w_n \in L$ such that $w = w_1 \cdots w_n$.
- In other words, given w, check if w is in L^*.
- We can construct an NFA for L^* and use the NFA recognition algorithm to check if there is a solution to the scanning problem.
- Problems with this approach:
  - The recognition algorithm just gives a yes or no answer. We want an actual decomposition of the string into tokens.
  - If the decomposition exists, it might not be unique. We need a consistent way to pick which decomposition to use.
- Example: if the language of tokens is "numbers with no leading zeroes", the word 4242 could be split up in many ways:
  - [4] [2] [4] [2]
  - [42] [42]
  - [4242]
- For a programming language, we probably want the last one.

Maximal Munch

- Given w, find words $w_1, \ldots, w_n \in L$ such that $w = w_1 \cdots w_n$.
- Maximal Munch Scanning: Given w, run the following algorithm, starting with $i = 1$:
  1. Let w_i be the longest prefix of w such that $w_i \in L$. If no prefix of w is in L, throw an error.
  2. Write $w = w_i x$. If x is empty, return the sequence of tokens $w_1, \ldots, w_i$. Otherwise, replace w with x, increment i by 1, and repeat the previous step.
- Example: Let $L = \{abb, abba, abbabba\}$ and $w = abbabbaabbaabba$.
  - First iteration: $w_1 = abbabba$, $x = abbaabba$.
  - Second iteration: $w_2 = abba$, $x = abba$.
  - Third iteration: $w_3 = abba$, $x = \varepsilon$.
  - The tokens returned are [$abbabba$] [$abba$] [$abba$].
- Example: Let $L = \{abb, abba, abbabba\}$ and $w = abbaabbabb$.
  - First iteration: $w_1 = abba$, $x = abbabb$.
  - Second iteration: $w_2 = abba$, $x = bb$.
  - Third iteration: an error is thrown.
- Notice in the above example, a valid scan exists: [$abba$] [abb] [abb].

Pros and Cons of Maximal Munch

- If there are multiple solutions to the scanning problem, Maximal Munch will pick a unique solution and return that one consistently.
- This means there is no ambiguity in the programming language specification, but it also means it can be hard to understand the specification. Instead of an intuitive explanation the specification is defined in terms of an algorithm.
- Always taking the longest prefix is often, but not always, the desired solution in programming languages.
- For example, if we have a long number with no spaces in it, we want it to be read as one long token instead of being split up into a bunch of smaller numbers.
- On the other hand, in older versions of `C++`, writing `list<list<int>>` is an error because the final >> gets scanned as an operator. You needed a space for it to scan correctly: `list<list<int> >`
- Even when a solution exists, Maximal Munch might fail to find the solution and produce an error.

Context-Free Languages

- Regular languages cannot handle an arbitrary amount of nesting, but nested structures are common in programming languages:
  - `1+(2+(3+(4+(...))))`
  - ```
    if(conditionOne) {
        if(conditionTwo) {
            if(conditionThree) {
                ...
            }
        }
    }
    ```
  - `procedureOne(procedureTwo(...),procedureThree(...))`
- Examples of context-free but non-regular languages:
  - Sequences of matching parentheses.
  - Arithmetic expressions with parentheses.
  - Words over $\{a, b\}$ where the number of occurrences of a equals the number of occurrences of b.
- The language of valid Lacs programs is not context-free (requires "context-sensitive" analysis for name resolution and type checking).
    ```

Context-Free Grammars

- Consider the following context-free grammar:

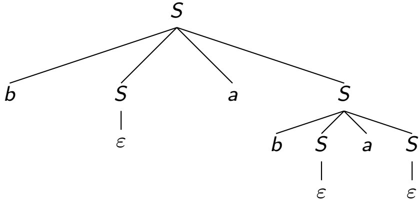
$$S \to \varepsilon \mid aSbS \mid bSaS$$

  - The terminal symbols are $\{a, b\}$.
  - The only non-terminal symbol is S.
  - There are three production rules: $\{S \to \varepsilon, S \to aSbS, S \to bSaS\}$.
  - The start symbol is S.
- By convention, the non-terminal symbol on the left hand side of the first line of the grammar is assumed to be the start symbol. If the start symbol is ever not specified, follow this convention.
- A derivation of a word w is a sequence of applications of production rules, starting from the start symbol and ending with a string of terminals equal to w.
- Example: Derivation of $baba$:

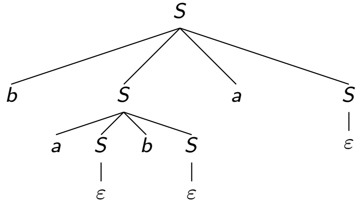
$$\underline{S} \underset{S \to bSaS}{\Rightarrow} b\underline{S}aS \underset{S \to \varepsilon}{\Rightarrow} ba\underline{S} \underset{S \to bSaS}{\Rightarrow} bab\underline{S}aS \underset{S \to \varepsilon}{\Rightarrow} baba\underline{S} \underset{S \to \varepsilon}{\Rightarrow} baba.$$

Parse Trees and Ambiguity

- Parse tree for *baba*:



- Another parse tree for *baba*:



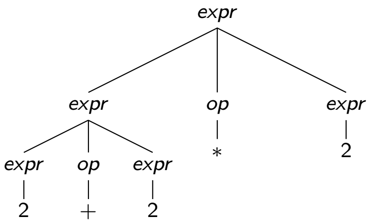
- A grammar is ambiguous if there exist two different parse trees for some word. Ambiguity is undesirable for programming languages!

Parsing and Ambiguity

- This grammar for simple arithmetic expressions is ambiguous:

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

- Our CYK parser can handle ambiguous grammars and will happily parse expressions with this grammar, returning some parse tree.
- But what if it returns this tree for $2 + 2 * 2$?

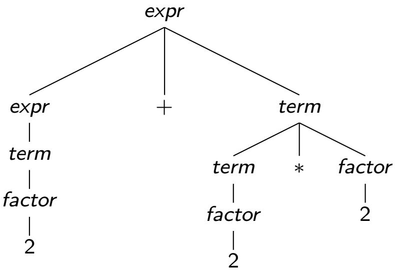


- If we evaluate the expression using this tree, the order of operations will be wrong!

- Solution (used by Lacs): Make multiple "levels" of expressions, with multiplication being at a deeper level than addition.

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

- The expression $2 + 2 * 2$ has a unique parse tree under this grammar with correct order of operations.



CYK Algorithm: Overview

- 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).
- Let's look at how R behaves on various inputs.

CYK Algorithm: Examples

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

$$S \to A \mid SA$$
$$A \to a$$

- The parse tree returned by $R(a, a)$ consists of a single node which just contains a.
- For $R(A, a)$, the parser will try the rule $A \to a$ and then construct a tree of the form:

$$\begin{array}{c} A \\ | \\ R(a, a) \end{array}$$

Which results in the tree:

$$\begin{array}{c} A \\ | \\ a \end{array}$$

CYK Algorithm: Examples

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

$$S \to A \mid SA$$
$$A \to a$$

- $R(S, a)$ results in the tree:

$$\begin{array}{c} S \\ | \\ A \\ | \\ a \end{array}$$

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:

$$\begin{array}{c} S \\ | \\ R(A, a) \end{array}$$

CYK Algorithm: Examples

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

$$S \to A \mid SA$$
$$A \to a$$

- For $R(aA, aa)$ we get the following two trees:

$$a \qquad\qquad \begin{array}{c} A \\ | \\ a \end{array}$$

  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 $Seq(a, R(A, a))$.

CYK Algorithm: Examples

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

$$S \to A \mid SA$$
$$A \to a$$

- For $R(SA, aa)$ we get the following two trees:

$$\begin{array}{c} S \\ | \\ A \\ | \\ a \end{array} \qquad\qquad\qquad \begin{array}{c} A \\ | \\ a \\ \\ \\ \end{array}$$

  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 into ε and aa, do recursive calls $R(S, \varepsilon)$ and $R(A, aa)$. This fails.
  - Split 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 $R(A, \varepsilon)$.

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

$$S \to A \mid SA$$
$$A \to a$$

- For $R(S, aa)$, we get the following tree:

$$\Tree [.S [.S [.A a]] [.A a]]$$

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 $R(SA, aa)$.

CYK Algorithm: Memoization

- This grammar is not only ambiguous, but the word $\{a\}$ has infinitely many parse trees.

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

- The CYK parser handles this using memoization: it remembers which inputs it has attempted to parse before to avoid getting into cycles.
- Here is what will happen when the recursive parsing function R is called with inputs $\alpha = S$ and $x = a$:
  1. $R(S, a)$ is called. The CYK parser makes note of the fact that it has seen the pair (S, a) in its memoization table.
  2. The parser tries the rule $S \to A$ and calls $R(A, a)$.
  3. The parser tries the rule $A \to S$. But it realizes it has seen the input (S, a) before so it does not recurse and avoids entering a cycle.
  4. The parser then tries the rule $A \to a$ and calls $R(a, a)$, which leads to a successful parse.

Comparing Different Parsers

	CYK	Earley	LR(k)	LL(k)
Time complexity	$O(n^3)$	$O(n^3)$ ambiguous $O(n^2)$ unambiguous $O(n)$ most LR(k)	$O(n)$	$O(n)$
Which grammars?	All	All	most practical unambiguous grammars	few practical grammars
Correct prefix property	No	Yes	Yes	Yes

- Here n is the length of the input word.
- The speed difference between CYK and Earley is exploited by the Marmoset tests to make sure you don't cheat.
- LR parsers can only handle unambiguous grammars, and not even all unambiguous grammars, but they work well in practice and are fast.
- A parser has the correct prefix property if it rejects invalid inputs as soon as possible – once the prefix it has read cannot be extended to a valid input anymore.
- This is useful for error reporting in programming languages.

Context-Sensitive Analysis

- Even after successfully scanning and parsing a program, there can still be errors, and we might not have all the information needed for code generation.
- The point of context-sensitive analysis is to:
  - Detect additional errors that are not caught by the scanner or parser.
  - Construct some data structures with useful information about the program to help with code generation.
- In Lacs, context-sensitive analysis consists of name resolution and type checking.
  - Name resolution: construct a symbol table for each procedure, containing all the variable and procedure names that are in the procedure's scope. Check that no undefined or out-of-scope variables are ever used, and check that there are no duplicate variable names.
  - Type checking: Determine the type of every expression in the program, while simultaneously enforcing type rules such as "the arguments of the addition operator must be integers" and "the left hand side and right hand side of an assignment must have the same type".

Name Resolution

- ```
  def main(a:Int,b:Int):Int = { f() }
  def f():Int = {
      def f():Int = { 0 }
      f()
  }
  ```
- This Lacs code compiles. There is no duplicate identifier error.
- This Lacs code returns 0. It does not go into an infinite recursion.
- To understand why, let's look at how the symbol table for the outer `f` is constructed.
- We start with the top level symbol table that contains all the outermost procedures in the program as a base.

Name	Information
`main`	procedure of type `(Int,Int)=>Int`
`f` (outer)	procedure of type `()=>Int`

- Note that every procedure in the program inherits from this top level symbol table. This is what enables us to call top level procedures.
  ```

Name Resolution

- ```
  def main(a:Int,b:Int):Int = { f() }
  def f():Int = {
      def f():Int = { 0 }
      f()
  }
  ```
- This Lacs code compiles. There is no duplicate identifier error.
- This Lacs code returns 0. It does not go into an infinite recursion.
- To understand why, let's look at how the symbol table for the outer `f` is constructed.
- Now we add all the names declared in the outer `f`. The only name is the inner `f`, which overrides the outer `f` because the name is the same.

Name	Information
`main`	procedure of type `(Int,Int)=>Int`
`f` (inner)	procedure of type `()=>Int`

- When outer `f` calls the procedure named `f`, it looks up `f` in the symbol table and finds inner `f`, so the result is that the program returns 0.
  ```

Type Checking

- ```
  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)
  }
  ```
- First type check the procedure `e`.
- Recursively, type check the procedure `d` nested in `e`.
- In this scope, `a` has type `(Int,Int)=>Int` and `b` and `c` are both `Int` type, so the call to `a` is correctly typed, and the return value's type matches the stated return type of `Int`.
- Now check the expressions in the body of `e`.
- There is only one, `d` of type `(Int)=>Int`, which matches the return type of `e`.
- Nested procedures `d` and `e` are correctly typed.
  ```

Type Checking

- ```
  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)
  }
  ```
- Now check the expressions in the body of `f`.
- For `d = e`, do variable `d` and procedure `e` have the same type? Yes.
- For `e(if(c>0) { a(c) } else { b(c) }, c)`, check parameters.
- For the if condition, `c` and `0` are both `Int` type.
- Since `a` and `b` have the same type, so do `a(c)` and `b(c)`, so the if and else clauses return values of the same type `(Int,Int)=>Int`.
- This type is the correct type for the first parameter of `e`. The second parameter is also correctly typed.
- The return type is the return type of `e`, which is `(Int)=>Int`.
  ```

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

Let's run garbage collection on the following heap and stack.

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	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

Stack
32
4
128
152
128
192
140
42

Start by scanning the stack. There are 4 pointers in this chunk.

Memory Management

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	12	192	12
132	1	196	1
136	168	200	168
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

Stack
32
4
128
152
128
192
140
42

Copy the chunk at 128 to the to-space.

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	-12	192	12
132	1	196	1
136	168	200	168
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

Stack
32
4
128
152
128
192
140
42

Negate the size to mark the chunk as copied.

Memory Management

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	-12	192	12
132	192	196	1
136	168	200	168
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

Stack
32
4
128
152
128
192
140
42

Put the new address of the copied chunk in the second word.

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	-12	192	12
132	192	196	1
136	168	200	168
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

Stack
32
4
192
152
128
192
140
42

Update the address of the copied chunk on the stack.

Memory Management

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	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

Stack
32
4
192
152
128
192
140
42

Copy the chunk at 152, mark it as copied, and leave the new address.

Memory Management

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	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

Stack
32
4
192
204
128
192
140
42

Update the address of the copied chunk on the stack.

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	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

Stack
32
4
192
204
192
192
140
42

Chunk at 128 already copied, so we just update the address.

Memory Management

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	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

Stack
32
4
192
204
192
192
140
42

Chunk at 192 is not in the from-space, so we leave it alone.

Memory Management

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	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

Stack
32
4
192
204
192
192
140
42

Ignore the last two variables because they are not pointers!

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	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

Stack
32
4
192
204
192
192
140
42

Now scan the to-space. This chunk has one pointer.

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	-12	192	12
132	192	196	1
136	168	200	168
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	220	24
160	128	224	2
164	168	228	152
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

Copy the chunk at 168, mark it as copied, and leave the new address.

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	-12	192	12
132	192	196	1
136	168	200	220
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	220	24
160	128	224	2
164	168	228	152
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

Update the address of the copied chunk.

Memory Management

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	-12	192	12
132	192	196	1
136	168	200	220
140	12	204	16
144	1	208	2
148	140	212	128
152	-16	216	168
156	204	220	24
160	128	224	2
164	168	228	152
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

Next chunk has two pointers.

Memory Management

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	-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	152
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

Both point to already-copied chunks, so we update the addresses.

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	-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	152
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

Next chunk has two pointers.

Memory Management

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	-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

One is a copied chunk. The other is outside the from-space.

Memory Management

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	-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

We are done because we have scanned all chunks in the to-space.

The End

Final Review Solutions:

`https://www.student.cs.uwaterloo.ca/~cs241e/current/FinalReviewSolutions.pdf`

(The link will be posted on Piazza and the website when I get home)

Good Luck :v)