SEG 2106 Summary Sheet

1 Software Development Processes

1.1 Black and White Box

The **black box** process gets the requirements for a project, then the team works on the project for a set period of time, and produces a product.

Thie **white box** process is similar to the black box process except for the stakeholders periodically check up on the project to provide **feedback**.

1.2 Models

The **waterfall model** sequentially does domain analysis, gets requirements, makes a design, then tests and finally deploys.

This is very easy and good under ideal circumstances, but in the real word this is rarely used since **circumstances are rarely ideal**. Often stakeholders do not effectively communicate requirements to developers, or the requirements change.

We can modify the waterfall by adding **feedback** into the loop at each stage, so if we need to change something, we do not have to redo the whole project. This is an **agile development process**.

2 Requirements Modeling

To get the requirements for a project, we need to analyse the domain the software will be used in. We need to understand the domain that the users are in so we can create software that works for the users.

We can make UML class diagrams for a system and all its parts.

2.1 Requirements

We need to interview the stakeholders to figure out what they want. These are usually lots of sentences that say in general non technical terms what the system needs to do.

A system will have lots of use cases. Each use case is a "thing" that the user wants to do. Such as "create text file" or "delete file".

We can create a use case diagram which is a diagram depicting all the use cases, and what user does that use case.

For each use case, we have the following information:

- Actors
- Summary
- Precondition (does the system have to be in a specific state to do this use case?)

- Main Sequence (Sequence of what the user does and how the system responds)
- Alternative Sequences (Basically modifications to the main sequence depending on other events.)
- Postcondition (What state the system is in after the use case.)

Ex.

Functional Requirements are things that the system is supposed to do.

Non Functional Requirements are more about how the system is supposed to do something. Such as the system "must complete action in less than 5 seconds" or the system "must store password as encrypted".

A **user story** is a more general form of a use case.

3 Behavioral Modeling

3.1 UML Activity Diagrams

This is a diagram that models behaviour of a system. They are good to model concurrent behaviour.

Ex.

3.2 UML State Machines

This is a high level diagram that models the states of a certain part of a system. It is good at working with a single object/part.

3.3 Petri Nets

This is a diagram that is good for modeling complex concurrent behaviour. They get very complex very fast though, so they are not great for large projects unless we have a large time budget.

A petri net works off of tokens. Tokens can trigger transitions that will cause tokens to move locations, add new tokens, or consume tokens.

A petri net is **bounded** if the number of tokens cannot exceed a finite amount. Conversely, a petri net is **unbounded** if *there exists at least one* way for a petri net to create an infinite number of tokens.

A petri net is **alive** if there is always at *least one transition* that can be fired from any point. Conversely, if there is any reachable combination where the system cannot do anything else (it dies), the petri net is not alive.

Ex.

3.4 Specification and Description Language (SDL)

SDL has 3 main hierarchical levels. System, Block, and Process.

A system has one or more blocks interconnected with other.

A process is a state machine that has states and transitions, kind of like a UML state machine.

4 Lexical Analysis

This is one part of a compiler. It checks that all the words in the program are **part of the language.**

For example, if the language is java, "print" is not a word, but "if" and "System.out" are words.

Regular Expressions are used to do this.

4.1 Alphabet

The **alphabet** are all the characters that we are working with. If a letter not in the alphabet is found, then the string is not accepted.

Ex. If the alphabet is a,b,c then:

"aaa" may be good

"aaabbbbcccc" may be good

"aaabbbcccddd" is NOT good since "d" is not in the alphabet.

4.2 Operations

Union is the or operator. A—B means A or B

Concatenation is 2 consecutive strings. AB means the string AB, not BA, AA, BB...

Kleene Closure is the set of all strings that can be created from 0 or more strings defined. (aa)* means kleene closure of (aa) which is "", "aa", "aaaa", "aaaaaa", but not "a", "ab", "aaa"...

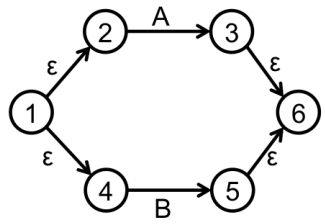
Positive closure is like kleene closure but excludes the empty string. So, (aa)+ is the positive closure of (aa) which is "aa", "aaaa", ... Note the exclusion of "".

4.3 NFA

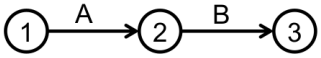
Non-deterministic Finite Automata (NFA) are a type of machine that model a regular expression. These have multiple states that are connected to each other with transitions. These transitions can be either ϵ (the empty string) or any value in the alphabet.

We use the following rules to convert a regular expression (regex) into an NFA:

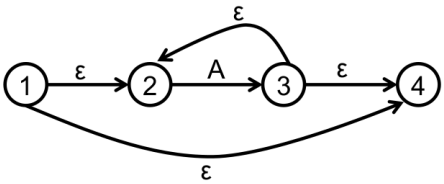
A | B



AB



A*



We can combine these together to model more complicated REGEX such as (a—b)* which is just the A* design, but instead of A, we substitute the A—B design.

Ex. Find the NFA of _____

4.4 DFA

NFA is easy to get, but it is hard for the computer to interpret it. This is where the Deterministic Finite Automata (DFA) comes into play. We create this using the **subset construction algorithm** on a NFA. In worst case scenario, we could have 2^n states where n is the number of states in NFA.

We have 2 functions.

ϵ-closure(S) is the state of the system that is reached by all epsilon transitions from state S.

move(S,a) will be the state the system is in after starting in state S and then taking the transition a.

Basically, we make a table with all the states of the system (up to 2^n) as rows, and colums for each letter of the alphabet. Then we perform move for each row (S), and each column (a)

Ex. Get the DFA of the following NFA through the subset construction algorithm:

5 Syntax Analysis

The parser is the part that takes in the tokens from scanner and checks that the grammar is correct.

We work with Context Free Grammars (CFG).

A CFG contains:

- Terminals - Tokens produced by lexical analyser (actual raw code we are working with)
- Non terminals - Variables that denote sets of strings
- Start Symbol - A Special non terminal which is where we start
- Productions - Specify the manner the terminals and non terminals can be combined (Grammar Rules)

Ex. This is an example of a grammar.

```
<expr> ::= <expr> <op> <expr> | id | num
<op> ::= + | − | * | /
```

We start with `<expr>` which can become either the terminals `id` or `num`, or can take on a non terminal of `<expr>␣<op>␣<expr>`.

Then the `<op>` nonterminal can take on any of the operators.

Note that this grammar is useful for variable additions, subtractions, etc, such as:

```
a + 1 // <expr> <op> <expr>
b − 5 // <expr> <op> <expr>
c * c + 1 // <expr> <op> (<expr> <op> <expr>)
```

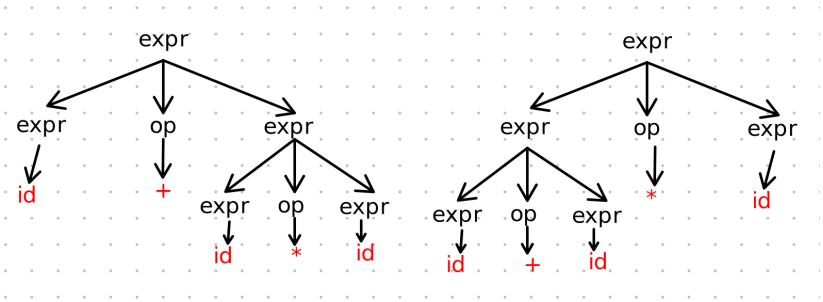
5.1 Parse Trees and Ambiguity

We can create a parse tree from a grammar to parse a certain string.

We call a grammar **ambiguous** if more than one parse tree can be created for a certain string.

Ex. Using the previous grammar provided, produce a parse tree for `id+id*id`.

We start with the starting symbol. Then we need to expand one of the `<expr>` nonterminals to get `<expr><op><expr>`, then web sub in `id` for each `<expr>`, and the corresponding operators.



> We can actually create at least **two** parse trees for this same string. So we know that this grammar is ambiguous as well (uh oh).

It is very hard to methodically eliminate ambiguity, and not something that is done in this course.

5.2 Left Recursion

This is where there is a rule such as $A \implies A\alpha|\beta$ where A is a nonterminal, and α is anything else.

This is a problem since the parser will expand A to get $A\alpha$. Then it will expand that A to get $A\alpha\alpha$. Then again to get $A\alpha\alpha\alpha$ and so on. It will get stuck in an infinite loop.

Whenever a rule like that exists, we need to remove it.

We fix this by doing:

```
A ::= bA'
A' ::= aA' | ϵ
```

instead of:

```
A ::= Aa | b
```

5.3 Left Factoring

If we have a grammar rule that says $A ::= Ba|Bc$, then we do not know where it should go. If we see a B, do we go to the first rule, or the second rule?

We can figure this out by backtracing if we choose the wrong rule, but this is inefficient.

To do this, we will change that rule into two rules of $A ::= BA'$, and $A' ::= a|b$.

5.4 FIRST and FOLLOW sets

These are two sets that need to be calculated for the computer to do syntax analysis.

Formally they are defined as:

1. FIRST(terminal) is {terminal}
2. If A→aα, and a is a terminal:
   {a} ∈ FIRST(A)
3. If A→Bα, and rule B→ε does **NOT** exist in the grammar:
   FIRST(B) ∈ FIRST(A)
4. If A→Bα, and rule B→ε **DOES** exist in the grammar:
   {(FIRST(B) - ε) ∪ FIRST(α)} ∈ FIRST(A)

1. {$} ∈ FOLLOW(S) → (where S is the starting symbol)
2. If A→ αB:
   FOLLOW(A) ∈ FOLLOW(B)
3. If A→αBγ, and γ→ε does **NOT** exist in the grammar:
   FIRST(γ) ∈ FOLLOW(B)
4. If A→αBγ, and γ→ε **DOES** exist in the grammar:
   {(FIRST(γ) - ε) ∪ FOLLOW(A)} ∈ FOLLOW(B)

These definitions are a bit hard to follow though.

Informally, the FIRST set contains the FIRST of all the nonterminals. The FIRST of a nonterminal is the **first terminal that will show up** in place of this nonterminal.

Informally the FOLLOW set contains the FOLLOW of all the nonterminals. The FOLLOW of a nonterminal is the **first terminal that appears after** this nonterminal.

Ex. Remove left factoring and left recursion from this grammar. Then create the first and follow sets.

```
<program> ::= {<statement_list>}
<statement_list> ::= <statement>;<statement_list>  | <statement>;
<statement> ::= call:  <procedure_call>  | compute:  <expression>
<procedure_call>::= id(<parameters>)
<parameters>::= <factor>,<parameters>  | <factor>
<expression> ::= id=<factor>+<factor>  | id=<factor>-<factor>  | id=<factor>
<factor> ::= id|num
```

To remove **left recursion**, we need to look for something like $A ::= A\alpha$. We do not have anything of the sort present.

To remove **left factoring**, we look for where a rule can derive more than 1 rule each with the same first item.

This is the case with `<statement_list>` since both derivations start with `<statement>`. Following the rules for removing left factoring we get:

```
<statement_list> ::= <statement>;<statement_prime>
<statement_prime> ::= <statement_list> | ϵ
```

We also need to do this with `<parameters>` and `<expression>` to get:

```
<program> ::= {<statement_list>}
<statement_list> ::= <statement>;<statement_list'>
<statement_list'> ::=<statement_list> | ϵ
<statement> ::= call:  <procedure_call>  | compute:  <expression>
<procedure_call>::= id(<parameters>)
<parameters> ::= <factor><parameters'>
<parameters'> ::= ,<parameters> | ϵ
<expression> ::= id=<factor><expression'>
<expression'> ::= +<factor> | -<factor> | ϵ
<factor> ::= id|num
```

Now to create the first set.

The FIRST for `<program>` would be just `{` since this is the first thing we see.

The FIRST for `<statement_list>` is more complex. The first thing we see is the first of `<statement>` which is `call` or `compute`.

We do this for each element. Now for the FOLLOW sets.

For `<program>` we see where `<program>` is found in the rules. It is never found. So the only thing that can come after it is the EOF (end of file) symbol `$`.

For `<statement_list>`, we see that this is found in the definition of `<program>`, and

<statement_list'>.

- Considering the <program> rule, what follows is }.
- Considering the <statement_list'>, well the last element is <statement_list>, so here we need to add the FOLLOW of <statement_list>' which is the FOLLOW of <statement_list> which means we can ignore this rule.

So now we also have the FOLLOW of <statement_list'> which is also }

Then for <statement> the element following it is only ;

We end up getting:

FIRST(program) = {	FOLLOW(program) = $
FIRST(statement_list) = call compute	FOLLOW(statement_list) = }
FIRST(statement_list') = call compute ϵ	FOLLOW(statement_list') = }
FIRST(statement) = call compute	FOLLOW(statement) = ;
FIRST(procedure_call) = id	FOLLOW(procedure_call) = ;
FIRST(parameters) = id num	FOLLOW(parameters) =)
FIRST(parameters') = , ϵ	FOLLOW(parameters') =)
FIRST(expression) = id	FOLLOW(expression) = ;
FIRST(expression') = + - ϵ	FOLLOW(expression') = ;
FIRST(factor) = id num	FOLLOW(factor) = ; , + -)

5.5 Parse Tables

A Parse table is a table where the rows are all of the non terminals, and the columns are all the terminals including the end of line character $.

For the first row, we go through the FIRST set of that nonterminal. Each terminal that exists, we fill out that box with the rule associated to the nonterminal/terminal combination.

If the FIRST set contains the empty string ϵ, then we also include the terminals from the FOLLOW set.

Ex. For the previous example, the parse table is below:

	id	num	call	compute	+	-	;	:	,	=	(	)	{	}
program	-	-	-	-	-	-	-	-	-	-	-	-	<program> -> {<statement_list>}	-
statement_list	-	-	<statement_list> -> <statement>; <statement_list'>	<statement_list> -> <statement>; <statement_list'>	-	-	-	-	-	-	-	-	-	-
statement_list'	-	-	<statement_list'> -> <statement_list>	<statement_list'> -> <statement_list>	-	-	-	-	-	-	-	-	-	<statement_list'> -> ϵ
statement	-	-	<statement> -> call: <procedure_call>	<statement> -> compute: <expression>	-	-	-	-	-	-	-	-	-	-
procedure_call	<procedure_call> -> id(<parameters>)	-	-	-	-	-	-	-	-	-	-	-	-	-
parameters	<parameters> -> <factor> <parameters'>	<parameters> -> <factor> <parameters'>	-	-	-	-	-	-	-	-	-	-	-	-
parameters'	-	-	-	-	-	-	-	-	<parameters'> -> , <parameters>	-	<parameters'> -> ϵ	-	-	-
expression	<expression> -> id=<factor> <expression'>	-	-	-	-	-	-	-	-	-	-	-	-	-
expression'	-	-	-	-	<expression'> -> +<factor>	<expression'> -> +<factor>	<expression'> -> ϵ	-	-	-	-	-	-	-
factor	<factor> -> id	<factor> -> num	-	-	-	-	-	-	-	-	-	-	-	-

5.6 LL1 Parsing

To actually do the LL1 Parsing, we make use of the **parse table.**

We have a stack that we work with.

1. Push the **start symbol** onto the stack.
2. Popping the top nonterminal on the stack, and the next terminal in the input stream, we then push the corresponding cell in the parse table.
3. **IF** the top of the stack has a terminal, **THEN** we check if this terminal corresponds to the head of the input stream.

   IF YES then we pop this element from the stack, as well as from the input stream and continue

   IF NO then the syntax does not follow the grammar and we have a problem. We **terminate the syntax analysis**
4. **Return to step 2 and repeat** until no terminals left in the input stream.

5.7 Error Recovery

There can be two types of errors when parsing:

1. There is a missing token (like missing semicolen)
2. There is an unexpected token (like extra semicolen)

The parser will try to skip symbols from the input until it gets back in sync, then it will skip one from the stack.

We are in sync when the current cell is a cell in the follow set.

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)={+, -, $}
FOLLOW(term')={+, -, $}
FOLLOW(factor)={*, /, +, -, $}
```

Grammar

⟨expr⟩ ::= ⟨term⟩⟨expr′⟩
⟨expr′⟩ ::= +⟨expr⟩ | −⟨expr⟩ | ε
⟨term⟩ ::= ⟨factor⟩⟨term′⟩
⟨term′⟩ ::= ∗⟨term⟩ | /⟨term⟩ | ε
⟨factor⟩ ::= num | id

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	(s)
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε (s)
term	<term>→ <factor><term'>	<term>→ <factor><term'>	(s)	(s)	-	-	(s)
term'	-	-	<term'>→ ε (s)	<term'>→ ε (s)	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε (s)
factor	factor→ num	factor → id	(s)	(s)	(s)	(s)	(s)

6 Concurrency

Concurrency means having **multiple threads** operating at the same time. It sounds simple, but it causes a lot of headaches.

The main problems are some methods only can be accessed by one thread at a time or it will cause problems. For example, if two threadas try to **write to a file at the same time**, *which file do we keep*? That depends on who writes first.

For example, if two threads open a file that just has a counter variable inside (say at 0), and they each increment it by 1, and then both write the file, then depending on the order in which this was done, the counter variable in the file could either have value of 1, or 2.

This is known as **competition synchronization**.

We also have **cooperation synchronization** which is where there is a shared resource that multiple tasks need access to such as a printer. Another good example is a producer/consumer example. If the consumer consumes faster than the producer produces, then it will need to wait a lot for the producer.

6.1 Semaphores

A semaphore is a **counter**. This counter typically is either 1 or 0 (allowes access to only one thread) but could be more (such as allowing access to only up to 3 threads simultaneously).

For a thread to access the shared resource, it will need to **acquire** the semaphore with `wait(semaphore)` and when it is done it will **release** the semaphore with `release(semaphore)`.

The `wait` method will check if the semaphore's counter is greater than 0. If so, it will decrement the counter, and allow the task to access the resource. If the counter is 0, then it places the task in a queue.

The `release` method will either increment the counter (if queue is empty) or dequeue the next task to have access to the resource.

6.2 Deadlock

Deadlock happens when threads get to an impass and cannot proceed ever.

This often happens when a thread (Thread 1) requires another thread (Thread 2) to relenquish access to a resource. But Thread 2 is waiting for Thread 1 to relenquish access to a different resource. So neither can proceed.

This can be fixed by a few methods:

- Eliminating one of the semaphores. (Why not turn both semaphores into one semaphore)
- Recovering from deadlock by for example implementing time limits. (A semaphore gets auto released after 5 seconds or something)

6.3 Monitors

A monitor is a set of routines that are protected by a mutual exclusion lock.

It is basically a very big semaphore that encompasses a lot of methods.

They are built in to the programming languages, so they are easy to implement and understand but they do not offer as much control as semaphores.

A monitor only allows **one task** to access **ANY** subroutines in the monitor at any given time.

Monitors have 2 methods. `wait(aQueue)` puts the current thread in a queue to get access to the monitor. `signal(aQueue)` tells the queue `aQueue` that it can let someone else into the monitor.

6.4 Atomic Variables

An atomic variable is a variable that all modifications are done in a **single step**. This ensures synchronization.

They are a class such as `AtomicInteger` or `AtomicBoolean`.

6.5 Java Concurrency

A class can be made into a thread either by extending the `Thread` class, or implementing the `Runnable` interface. Both of these require the `public␣void␣run()` method to be implemented.

A `ThreadGroup` is a container that holds multiple threads. It lets us stop all threads, or find out how many threads are currently running.

When threads are interrupted, they have a flag set to `true` and they throw an `InterruptedException`.

The `synchronized` keyword for a method means that that method can only have one thread executint that method at a time. If there are multiple `synchronized` methods in a class, only **one** can be executed at a given time. This is called an **intrinsic lock** (monitor).

```
public synchronized int (getCount()) {
        counter++;
        return counter;
}
```

We can also use `synchronized` for a single block of code such as:

```
synchronized (lock) {a = 1} //synchronized variable assignment using a lock

//this cannot execute while the previous line is executing since same lock
synchronized (lock) {a = 2} //also synchronized using the same lock

synchronized (lock2) { //using a different lock
        b = 3
}
```

Cooperation synchronization is implemented using the `notify()` and `wait()` methods. These are used inside synchronized methods.

```
public synchronized int fetch() {
        while (buffer = 0) //while the buffer is empty
                wait();
        }
        //do some stuff
        buffer--;
        notify(); //let someone else know that there is space in the buffer to
            add to
}
```

We can also use something like `reentrantLock()`. This gives us a lock that we can lock and unlock. When it is locked, no one else can unlock it.

```
public class ------ {
        private final Lock lock = new ReentrantLock(); //creates lock
        public void test() {
                lock.lock(); //now no one else can unlock the lock
                try{ /*do some stuff*/}
                finally {lock.unlock;} //we ALWAYS need to unlock or deadlock
                   may occur
        }
}
```

We can also create conditions and use `condition.await()` to wait, and `condition.signal()` to signal threads waiting on that condition to wake up.