Project Information:

Our project name is Heap Heap Hooray, and our project will be to develop a garbage collector for the MiniJava compiler. This project will require us to deep-dive into garbage collector implementations, understanding and learning from them to write one that caters to the project's goals.

Our team consists of Tyler Gutowski (tgutowski2020@my.fit.edu), and Trevor Schiff (tschiff2020@my.fit.edu). Our advisor and client is Dr. Ryan Stansifer (ryan@fit.edu), a compiler researcher who has a deep understanding of the MiniJava compiler and Java runtime.

Progress Matrix:

Task	**Progress**	**Tyler**	**Trevor**
1. Comprehensive analysis of the Garbage Collector Handbook to grasp terminology, techniques, and algorithms	100%	0.5	0.5
2. Evaluation of the pros and cons of various garbage-collection strategies	100%	0.5	0.5
3. Examination of open-source garbage-collection projects for enhanced understanding	100%	0.0	1.0
4. Defining the objectives and scope of the project	100%	0.2	0.8
5. Determining the vital metrics to be measured	100%	1.0	0.0

6. Assessment of the project's feasibility considering the current skill sets	100%	0.5	0.5
7. Identification of prospective challenges and mitigation strategies	100%	0.0	1.0
8. Selection of the suitable garbage collection algorithm	100%	0.75	0.25
9. Development of high-level design outlining the main components of the garbage collector	100%	0.8	0.2
10. Crafting detailed design documents elucidating the data structures and algorithms to be employed	100%	0.0	1.0

Discussion:

1. We've begun reading the Garbage Collector Handbook, primarily reviewing the most important information, such as: jargon, basic garbage collection algorithms and generational vs incremental collection.
2.

	Pros	Cons
Reference Counting	● Simple to implement ● Memory is reclaimed in "real-time", and instantly.	● The counter itself takes up memory. ● Cannot detect cyclical references. ● Reclaiming memory is

		costly, which means that… ● Stack variables are inefficient.
Mark-Sweep	● Doesn't add auxiliary support to memory operations which means… ● GC operations only occur in GC cycles.	● All memory must be searched during GC cycle ● Fragments memory.
Copying	● Simple to implement. ● Defragments memory.	● Only a portion of the given memory is usable.
Generational	● Young generation is collected frequently, meaning… ● GC operations are fast.	● Tuning required. ● Complex. ● Only a portion of the given memory is usable.
Incremental	● GC operations are short and often, meaning that… ● No GC cycle stutters.	● GC is slower. ● Complex.
Concurrent/Parallel	● GC runs in concurrence with incremental, meaning that… ● No GC cycle stutters	● Multi-threaded support. ● Not practical.

3. The project, Oilpan has some good advice and code snippets. This project uses marking, sweeping, and concurrence to achieve their results. Maybe a combination of algorithms would be the best option for what we are trying to accomplish.
4. Our primary goal is to get any sort of garbage collection implemented. A simple reference-counter will suffice. Our secondary goal is to implement more than one type of garbage collection, if possible. Perhaps have multiple different branches of the project, one for each algorithm, or even just run the different algorithms using parameters. Our tertiary goal is to make the garbage collector less abstracted, and attempt to visualize how the memory is being used to the user.
5. The vital metrics for the project are:

- Throughput
    - Allocations per second
    - Reclaims per second
- Pause Times
    - Maximum pause time
    - Average pause time
    - Pause time frequency
- Memory Overhead
    - Total memory allocated
    - Total memory available
- CPU Overhead
    - Resources used by application
    - Resources used by garbage collector

6. This project is feasible given our skill sets. Although we still do not have a complete understanding on garbage collection, we know enough to begin working. Trevor's understanding of compilers will be the most vital knowledge for this project.
7. Cyclic references pose a fundamental problem to the traditional reference counting strategy. We will need to decide how to best implement cyclic reference checking in order to work around these reference count deadlocks. Additionally, we must decide how to split the implementation between compiler integration and runtime integration. Lastly, we need to determine which method of freeing memory is best for our situation: freeing objects as soon as they become unreachable, or adding those objects to a "zero count list" where they can later be collected en masse.
8. We are going to start by implementing the simplest algorithm: Reference Counting. Then, once we get the foundations implemented, we will begin implementing the other algorithms as parameters, if possible.
9. Depending on the algorithms used, different components will be needed. Here are the components for the two primary algorithms: reference counting, and mark-and-sweep:
    - Reference Counting
        - Reference count field for each object
        - Initialize the objects
        - Increment/Decrement
        - Reclamation
    - Mark-and-Sweep
        - Save root references
        - Mark
            - Traverse the tree
            - Mark all objects in use
        - Sweep

        - Traverse the heap
        - Free all non-marked objects
        - Reset marks

10. Our current plans for the system design are illustrated in our team's Design Document.

Contributions:

Tyler Gutowski: I dove into the Garbage Collector Handbook to get a good grip on the basics. I checked out different garbage collection methods to see what might work best for us and set out the main goals and scope of our work. It was important to decide on the metrics to track our progress. I also picked out a garbage collection algorithm and sketched out a basic design for our collector. We're going to be starting small, but I expect that we'll get this implemented without much issue, so hopefully we'll be able to implement multiple algorithms.

Trevor Schiff: I looked at some open-source garbage collection projects to learn from them. Oilpan was a useful project that highlighted that using a combination of multiple garbage collection techniques might be the best option. Although Tyler and I researched together, I zeroed in on the details. I started thinking about challenges we might run into and how to handle them. I also put together some detailed plans about the structures and steps we'd use. Given my experience with compilers, I think I have some useful insights, especially when it comes to the next milestone.

Next Milestone:

Task	**Trevor**	**Tyler**
Reference counter creation	50	50
Reference counter referencing	50	50
Reference counter dereferencing	50	50
Add cyclic reference checks	50	50
Integrate with compiler	75	25
Performance analysis	0	100

Discussion:

In our upcoming milestone, we've planned a collaborative approach to tackle the core components of garbage collection. We're dividing responsibilities evenly for the creation, referencing, and dereferencing of the reference counter, both of us will address the addition of checks for cyclic references. While Trevor will spearhead the integration of the garbage collection mechanism into the mini-java compiler, the responsibility of performance analysis will fall squarely on Tyler. With our combined efforts, we're optimistic about the progression and outcomes for this phase of the project.

Client Meetings:

See Faculty Advisor Meetings below

Client Feedback:

See Faculty Advisor Feedback below

Faculty Advisor Meetings:

a

Faculty Advisor Feedback:

a

Approval from Faculty Advisor:

"I have discussed with the team and approve this project plan. I will evaluate the progress and assign a grade for each of the three milestones."

Signature: ______________________________ Date: _______

Evaluation by Faculty Advisor:

Tyler Gutowski	0	1	2	3	4	5	5.5	6	6.5	7	7.5	8	8.5	9	9.5	10
Trevor Schiff	0	1	2	3	4	5	5.5	6	6.5	7	7.5	8	8.5	9	9.5	10

Signature: _________________________________ Date: ________