Compiler Construction Project

TinyGame: A Mini Language Compiler

CS4031 - Fall 2025

Ahmed Yoshay (22K-4396) Musaddiq Kamal (22K-4432)
Anas Ahmed (22k-4371) Abdul Wadood (22K-5141)
Huzaifa Azam (22K-5078)

Project Overview

We designed **TinyGame**, a domain-specific language for 2D game scripting supporting entity declarations, property management, movement commands, conditionals, and arithmetic. Our compiler implements all six compilation phases from lexical analysis through code execution via an interpreter.

What We Learned

Lexical Analysis: We learned DFA construction for token recognition, handling two-character operators (==, !=), and distinguishing keywords from identifiers using lookup tables. Managing whitespace and implementing maximal munch taught us scanner state management fundamentals.

Syntax Analysis: Building a recursive descent parser clarified grammar-to-code relationships. We constructed ASTs, handled operator precedence, and generated meaningful error messages. Eliminating left recursion and ensuring grammar unambiguity was challenging but educational.

Semantic Analysis: Implementing two-level symbol tables (global entities, entity properties) taught us variable tracking and type enforcement. We performed type checking, detected undeclared variables, and validated property access patterns.

Intermediate Code Generation: Converting ASTs to three-address code revealed how high-level constructs decompose into low-level operations. We managed temporary variables, labels for control flow, and systematic transformation of nested conditionals into jump instructions.

Optimization: Implementing constant folding, algebraic simplification, copy propagation, and dead code elimination achieved 54% instruction reduction in test cases. Live variable analysis taught us to recognize optimization opportunities systematically.

Code Generation: Building an interpreter demonstrated runtime environment management, entity state handling, control flow implementation, and output generation. Executing our language on real programs was highly rewarding.

Challenges Encountered

- Property access semantic checking required careful entity-scoped symbol table design
- Ensuring correct operator precedence in parsing and TAC generation needed systematic grammar rules
- Nested conditionals demanded proper label management for jump targets
- Optimizations had to preserve semantics, especially for property assignments with side effects

What We Would Improve

Language Features: Add loops (while/for), user-defined functions for code reuse, multiple data types (strings, booleans), and else clauses for complete conditionals.

Error Handling: Implement error recovery to continue parsing after errors, add descriptive messages with suggestions for common mistakes.

Optimization: Add loop optimization, register allocation, cross-block common subexpression elimination, and peephole optimization for pattern-based improvements.

Code Generation: Target actual assembly (x86/ARM) or bytecode (JVM), implement runtime with garbage collection, add debugging support with breakpoints.

Testing: Develop comprehensive unit tests per phase, implement fuzz testing for edge cases, add integration tests for complex programs.

Key Takeaways

This project transformed our understanding of programming language implementation. We now appreciate the complexity behind simple statements and the engineering required for efficient compilation. The systematic decomposition through multiple phases exemplifies good software design. Most importantly, we learned that compiler construction requires both theoretical knowledge (automata, grammars, type systems) and practical skills (data structures, algorithms, testing). Building a working compiler gave us confidence to tackle complex systems and deeper respect for development tools.