VIPER

An amalgamation of our favorite programming quirks

Tommy Gomez, Trey Gilliland, Mustafa Eyeceoz,
Matthew Ottomano, Raghav Mecheri

AN OVERVIEW

WHY VIPER?

- Tradeoff: compiled vs interpreted
- Finding the perfect compromise for speed vs convenience

WHAT DID WE WANT?

- Top level executed code
- In built data structures
- Static typing for both speed and eliminating errors

VIPER

- Strongly, statically, typed
- Library support for data-structures
- Support for ternaries/guard expresions, loop iterators + syntactic sugar
- Support for arrow functions
- Restrictive scoping to allow for namespace conflicts

VIPER'S STRUCTURE

STRUCTURE

- The body of a Viper program consists of statements, and functions
- You can write code outside main!
- Globals are not accessible within functions, preventing Python-esque namespace conflicts

FUNCTION CALLS

- Support for function definitions and function calls
- Support for arrow functions

CONTROL FLOW/ITERATION

- Support for if statements, as well as guards and ternary operators
- Supports for standard looping as well as iterators (for...in)
- Support for skip (continue in Java) and abort (break in Java)

SCOPING

- Viper eliminates the global scope from within function bodies, eliminating the ambiguity that comes with global variables
- Support for function overloading

TYPE SYSTEM

SUPPORTED TYPES

int, float, bool, char, string, nah

Datatype	Memory
int	4 bytes
float	4 bytes
bool	1 byte
char	1 bytes
string	n/a
nah	n/a

ADDITIONAL DATA STRUCTURES

lists, dictionaries

VIPER'S STANDARD LIBRARY

- Viper uses the Viper C standard library as an API to enable in-memory data structures
- Support for lists, dicts, math functions, and type-casting
- Built-in functions like print

- Viper's lists are implemented as arraylists, and Viper's dicts are implemented as lookup tables that allow for for nested dictionaries as well
- Support for in-built in type-specific functions like len, contains, append, etc

NOTABLE FEATURES

GUARD EXPRESSIONS

```
int y = 0;

int test = ??
  y < 0 : 0
| y > 0 : 10
?? 5;
```

ITERATORS

```
int[] arr = [1,2,3,4];
for(int f in arr) {
  print(f)
}
```

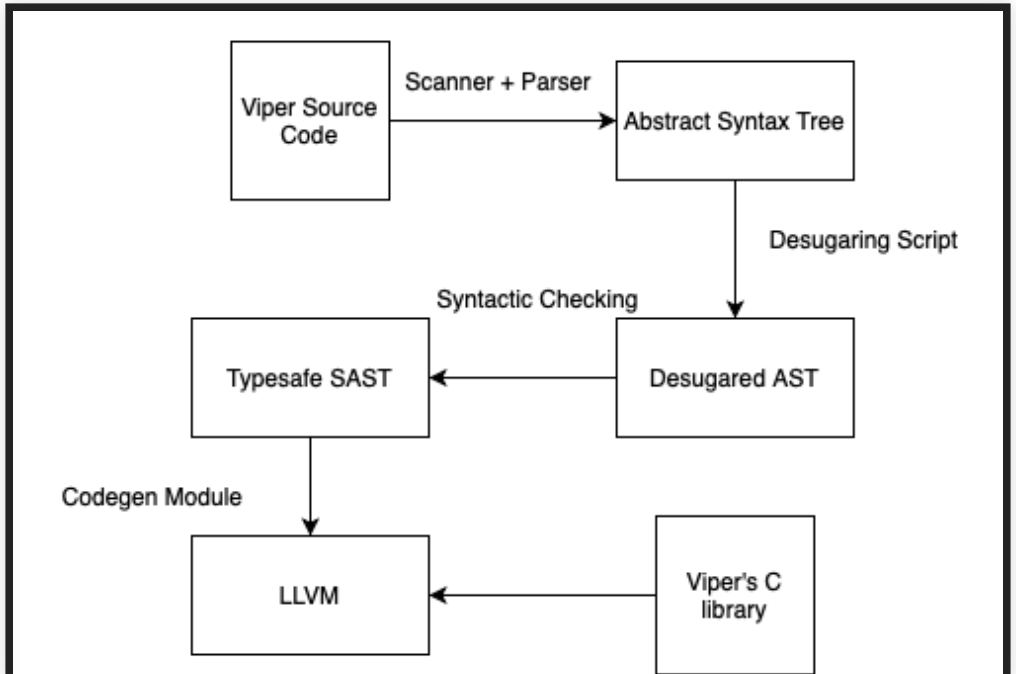
TERNARY OPERATORS

```
int val = 10;
int positive = val < 0 ? 0 : val
```

ARROW FUNCTIONS

```
int func add(int x, int y) => x + y;
```

COMPILER ARCHITECTURE


Viper Source Code
Scanner + Parser
Abstract Syntax Tree
Desugaring Script
Syntactic Checking
Typesafe SAST
Desugared AST
Codegen Module
LLVM
Viper's C library

HANDLING SYNTACTIC SUGAR

Viper has a lot of features that we were able to reduce

to just syntactic sugar, to make the work easier when we need to generate LLVM

OVERVIEW

Viper's desugar module runs via two passes, in order to break down more complex expressions before Codegen

OVERVIEW: FIRST PASS

- All of Viper's various for loops are reduced to while loops
- Arrow Functions are reduced to regular function calls

- Guard expressions are reduced to nested ternaries
- Attribute calls are decomposed to regular function calls [a.foo(b) -> foo(a, b)]

OVERIVEW: SECOND PASS

Viper's second pass serves only to convert all types of ternary operators into nested if statements

- Iterate through all the statements in a file
- Detect instances of expresions that contain ternary operators

- Insert a replacement if statement coupled with an assignment operator at the statement right before the

statement that contains the ternary, via a PretendBlock

- Viper uses PretendBlocks to insert blocks for which a reduced scope is not required - the SAST module ignores the change in scope

BUILDING THE SAST

HIGHLIGHTS

- Checks for duplicate and nah declarations
- Allows for function overloading
- Global statements

- Declarations inside of loops
- Entirely complete representation of anything that the AST throws at it

CHALLENGES FACED

The scoping for Viper is difficult to semantically check since we allow declarations inside of loops. Each time we enter a loop, we need to scan the declarations to

add to the symbol table incase they are called in the loop. Using symbol table parents allow us to keep track of declarations outside of the current scope and throws away deeper scopes once the program leaves a loop.

CODEGEN

HIGHLIGHTS

- Once I understood the workflow and how to use our C standard library functions I covered ground very quickly
- Efficient implementation for operations like Incr, Decr, OpAssigns

- Pointer Casting written in bindings for SAccess and other expressions
- Skip/Abort implemented!

CHALLENGES FACED

- Majority of codegen work done very late in the process due to needing types of expressions from SAST
- OCAML LLVM Documentation is very sparse

- Figuring out the LLVM workflow (write program -> compile to llvm with clang -> lookup instructions in documentation -> build similar instructions using OCaml bindings)

FUTURE WORK

- Allowing for imports of files/modules
- Support for other C-based libraries (numpy, tensorflow)
- Improve limited casting functionality/support
- Fixing all the bugs that keep cropping up

DEMO PROGRAMS