**Language Design Proposal**

**Student Name(s):** Ben Burrill

**Language Name:** Halt is Defeat

**Compiler Implementation Language and Reasoning:** Python. I am quite familiar with Python, and because I have already written the Sphinx emulator in Python, using Python for Halt is Defeat probably will make testing somewhat easier to do and might allow more interoperability between the compiler and emulator in other ways too.

**Target Language:** Sphinx Assembly

**Language Description:** A simple imperative language with structured features that make use of the predictive powers of the Sphinx architecture.

**Key Features:** First-order functions, Function overloading, Arrays, and Predictive defeat avoidance using try/undo with nested preemption.

**Planned Restrictions:** There will be no capability for dynamic memory allocation. Since “halt is defeat”, you are not able to write programs that terminate (all programs are guaranteed to end in an infinite loop), and there are several restrictions in how try blocks and function calls can be nested. Optimization will be very minimal. It is certainly not intended for any practical use, as the Sphinx emulator is itself impractical.

**Suggested Scoring and Justification:**

* **Lexer**: 5% The lexer should be relatively simple, there will be strings, but no string interpolation or anything too complicated.
* **Parser**: 10% There are lots of restrictions in how different blocks can be nested, which will probably complicate the grammar somewhat, but hopefully not too much.
* **Typechecker**: 15% Type system will be fairly minimal, but I want to have function overloading, and I might end up wanting some additional complexity down the line.
* **Code Generator:** 50% Compiling to a very weird low-level target.

**Overview:**

Although the jump instruction in Sphinx is prophetic, it can only give information about *what* will happen in the future (halting), not when or why. When writing modular code that predicts the future, you need some guarantees that the halt Sphinx is predicting is actually coming from the particular block of code you’re interested in. This can be accomplished by ensuring a path to an eventual infinite loop outside of the block of code which you are testing, so Sphinx will only predict a halt if that particular block would halt.

For this purpose, Halt is Defeat has “you-functions” with special calling restrictions so that returning from a you-function is guaranteed to eventually lead to an infinite loop.

You functions are prefixed with an @.

The entry point of the program is the you-function @is\_you. When the @is\_you function returns, the program enters into the trivial infinite loop known as “win”.

Essentially, you-functions determine the “real” path of execution, maintaining a link back to the safety of @is\_you. They are the only functions that can use try/undo, thanks to the guarantee that returning from a you-function leads back to the “real” path of execution which will eventually lead to win (or some other infinite loop at least).

To maintain this, you-functions can **only** be called directly from within other you-functions (and not within try blocks, since those represent a code path that could lead to defeat).

Other functions

There are also two other kinds of functions, which allow writing reusable code without requiring that it only be used in a “safe” context.

**Defeat functions** are functions which may cause defeat themselves. They can only be used in try blocks and other defeat functions. The built-in defeat function !is\_defeat() is implemented as an unconditional halt in Sphinx, and is ultimately the origin of all defeat.

Defeat functions are prefixed by ! (The logical negation operator in Halt is Defeat is not, just like in Python, and like the analogous word in the completely unrelated Baba is You).

**Ordinary functions** can be used from anywhere. They are basically like functions in any other language. They can’t predict the future and they can’t cause defeat.

Ordinary functions have no prefix.

I intend for some built-in ordinary functions like print to be implemented in Python, since having a bunch of loops implemented in Sphinx can get expensive. This will require some changes to the emulator.

Try blocks

As mentioned earlier, try blocks are allowed within you-functions, and allow a block of code to be avoided if it would lead to defeat. If running the try block would lead to defeat, the attached undo block is run instead.

Try blocks can have preempt blocks nested inside them. Preempt blocks get run when the containing try block would lead to defeat (ie if undo would get run) if the preempt block were skipped.

Preempt blocks are a bit weird in that they depend on what the surrounding code is going to do, in a way that sort of resembles what we were trying to avoid with the you-functions. But this weirdness is limited only to the try block which they are contained within, and they can be very useful / required for some prediction-based algorithms.

See the examples for more information about try/preempt/undo.

Summary of what’s allowed in different blocks

*(Unless otherwise stated, blocks preserve the rules of the block they are contained by)*

You functions:

* Function calls: ordinary functions, you functions
* Try blocks:
  + Function calls: ordinary functions, defeat functions
  + Preempt blocks
  + Conventional control blocks (while, if, etc)
* Undo block after try
* Conventional control blocks (while, if, etc)

Defeat functions:

* Function calls: ordinary functions, defeat functions
* Conventional control blocks (while, if, etc)

Ordinary functions:

* Function calls: ordinary functions
* Conventional control blocks (while, if, etc)

Recursion

All functions can be made recursive, but you-functions cannot make recursive calls within try blocks (as it is never permitted to call you-functions within try blocks). See calling convention at end of document for how function calls (including recursion) would work.

As far as the Sphinx emulator is concerned, without tail-call optimization, infinite recursion would not itself be detected as an infinite loop (it never repeats a state). This is accurate, since eventually “infinite” recursion will overflow a finite stack, leading to some other behavior, whether that may be halting, the emulator crashing, or in the case of Halt is Defeat, an actual infinite loop--see next section. None of that is particularly relevant to writing Halt is Defeat programs though.

Unrecoverable errors

When an error is encountered, such as a stack overflow or indexing an array out of bounds, the program produces a flag and enters into an infinite loop, much as it would upon returning from @is\_you. This effectively ends execution, and the path to the error will always be taken if encountered in virtual execution, since it will never lead to defeat. This means that errors are always reported, even if they occurred within blocks which might have otherwise not “really” been executed.

**Examples:**

Primality test

Defines a defeat-function, !composite\_is\_defeat which is defeat when the argument is composite, which is then used to define the boolean you-function @prime, which *immediately* returns false for composites. Uses built-in functions isqrt and println.

This is similar to examples/prime.s in Sphinx

void !composite\_is\_defeat(int n) {

for (int test = 2; test <= isqrt(n); test += 1) {

// I'll probably have !truth\_is\_defeat(bool cond) to simplify code like this

if (n % test == 0) {

!is\_defeat();

}

}

}

bool @prime(int n) {

// You could also put the loop from !composite\_is\_defeat directly inside the try

try {

println("It will be prime"); // Let the user know in advance

!composite\_is\_defeat(n);

return true;

} undo {

return false;

}

}

void @is\_you() {

// The entry point of the program

const int val = 100;

if (@prime(val)) {

println("It is prime");

} else {

println("It is composite");

}

}

Maximum value

This you-function finds the maximum value in an array, and makes use of preempt to return early as soon as the maximum value is encountered.

Actually, I think you could write @max using only try/undo in a loop and no preempt, though preempt makes it cleaner. An example which I think *requires* preempt is the sphinxfuck interpreter. In any case, this example gives some idea of how preempt works.

This is similar to examples/max.s in Sphinx

int @max(const int[] arr) {

try {

int max\_val = arr[0];

for (int i = 1; i < arr.length; i += 1) {

if (arr[i] > max\_val) {

max\_val = arr[i];

preempt {

// There are no larger values, so return early.

return max\_val;

}

}

}

// Reaching the end of the array without returning is defeat

!is\_defeat();

} undo {

// First item was the max, return it

// Probably we'd actually need to put this outside the undo,

// since otherwise it would be hard for the compiler to tell if

// we could get to the end of the function without returning.

return arr[0];

}

}

Abbreviated sphinxfuck interpreter

This is similar to examples/sphinxfuck.s in Sphinx

void @sphinxfuck(string sf, byte[] mem, string input) {

try {

int pc = 0, ip = 0, dp = 0;

while (true) {

if (sf[pc] == ...) {

... // Boring instructions

} else if (sf[pc] == '!' and not mem[dp]) {

!is\_defeat();

} else if (sf[pc] == '?' and mem[dp]) {

!is\_defeat();

} else if (sf[pc] == '[') {

preempt {

pc = find\_matching\_label\_ahead(sf, pc); // Helper to find ')'

}

} else if (sf[pc] == ']') {

preempt {

pc = find\_matching\_label\_behind(sf, pc); // Helper to find '('

}

}

pc += 1;

}

} undo {

// Unavoidable halt – ideally should run in alternate mode, chasing halt

}

}

**Syntax:**

NOTE: The block nesting restrictions described in “Summary of what’s allowed in different blocks” is not reflected in this grammar in order to avoid complicating things too much.

IDENT is a basic (unprefixed) identifier

INTEGER is an integer literal

STRING is a string literal

CHAR is a character/byte literal

SCALAR\_TYPE ::= `int` | `bool` | `byte` | `string`

Note on strings: I’ll probably implement it so that all strings must be declared as literals. So they’d always just be pointers to wherever in the const section the literal string they refer to got put.

TYPE ::= SCALAR\_TYPE | TYPE `[` `]`

RTYPE ::= SCALAR\_TYPE | `void`

You can’t return arrays

VTYPE ::= [`const`] TYPE

VDECL ::= VTYPE IDENT `=` EXPR | SCALAR\_TYPE IDENT (`[` EXPR `]`)+

Normal declaration must be initialized (variable length array declaration is not).

See end of document for more info about array declarations.

ARG\_SIG\_LIST ::= `(` [(VTYPE IDENT `,`)\* VTYPE IDENT] `)`

ARG\_LIST ::= `(` [(EXPR `,`)\* EXPR] `)`

FUNC ::= RTYPE IDENT ARG\_SIG\_LIST `{` STATEMENT\* `}`

YOU\_FUNC ::= RTYPE `@`IDENT ARG\_SIG\_LIST `{` STATMENT\* `}`

DEF\_FUNC ::= RTYPE `!`IDENT ARG\_SIG\_LIST `{` STATEMENT\* `}`

FUNC\_CALL ::= IDENT ARG\_LIST

YOU\_FUNC\_CALL ::= `@`IDENT ARG\_LIST

DEF\_FUNC\_CALL ::= `!`IDENT ARG\_LIST

BIN\_OP ::= `+` | `-` | `\*` | `/` | `%` | `==` | `<` | `>` | `<=` | `>=` | `!=` | `or` | `and`

UN\_OP ::= `+` | `-` | `not`

LOOKUP ::= IDENT | EXPR `[` EXPR `]`

INC\_ASSIGN ::= `+=` | `-=` | `\*=` | `/=` | `%=`

ASSIGN ::= LOOKUP `=` EXPR | LOOKUP INC\_ASSIGN EXPR

Assignments are not expressions.

ARRAY ::= `[` [(EXPR `,`)\* EXPR] `]`

EXPR ::= `(` EXPR `)` | UN\_OP EXPR | EXPR BIN\_OP EXPR | YOU\_FUNC\_CALL | DEF\_FUNC\_CALL | FUNC\_CALL | LOOKUP | INTEGER | STRING+ | CHAR | ARRAY | `true` | `false` | LOOKUP `.` `length` | EXPR `is` TYPE

Arrays and strings have an unassignable length attribute.

Adjacent strings are concatenated.

PLAIN\_STMT ::= EXPR | ASSIGN

LINE\_STMT ::= PLAIN\_STMT | VDECL

STATEMENT ::= BLOCK | [LINE\_STMT] `;` | `break` `;` | `continue` `;` | `return` [EXPR] `;`

BLOCK ::= `{` STATEMENT\* `}` | IF\_BLOCK | FOR\_BLOCK | WHILE\_BLOCK | TRY\_BLOCK | PREEMPT\_BLOCK

IF\_BLOCK ::= `if` `(` EXPR `)` BLOCK [`else` BLOCK]

WHILE\_BLOCK ::= `while` `(` EXPR `)` BLOCK

FOR\_BLOCK ::= `for` `(` [LINE\_STMT] `;` [EXPR] `;` [PLAIN\_STMT] `)` BLOCK

TRY\_BLOCK ::= `try` BLOCK `undo` BLOCK

PREEMPT\_BLOCK ::= `preempt` BLOCK

Note that for all these blocks I’m purposely defining them in a way so you can do stuff like preempt if (x) {}, but not preempt f(x);

GLOBAL\_STMT ::= VDECL | FUNC | YOU\_FUNC | DEF\_FUNC

PROGRAM ::= GLOBAL\_STMT\*

**Calling convention:**

The caller pushes a return address onto the stack, followed by arguments. The function pops everything off the stack when it returns, and then (for non-void functions) pushes a return value to the stack. Basically, it looks like this from caller’s perspective:

; result = function(42)

sws [sp], end\_call

add [sp], [sp], 1w

sws [sp], 42

add [sp], [sp], 1w

j function

halt

end\_call:

sub [sp], [sp], 1w

lws [result], [sp]

**Arrays and strings:**

There are two ways to declare arrays: int[] X = [1, 2, 3]; and int X[3];

Arrays cannot be reassigned after declaration, they can only be modified (assuming they aren’t const).

Arrays cannot be returned by functions, so you need to pass an output parameter.

The grammar also currently allows multi-dimensional arrays. I’m not sure I actually want to implement this, but if I do, I think I’d want these to be rectangular rather than reference based.

Array references might be implemented as views which hold the length/shape of the array and a pointer to the start of it. I think the only advantage currently is that *if I do* implement multi-dimensional arrays, and they are rectangular, it would neatly allow you to use a row of a multi-dimensional array as a reference, but potentially this could also allow for other kinds of slicing too (but that’s not specified in the grammar and almost certainly will not be implemented)

Strings act like an indirect reference to a byte array. The distinction to byte arrays is that non-const string variables can be reassigned (since they are just references to a string stored in const), and mutation of the underlying bytes is never permitted, though you can do lookups just like an array. Also because they are references, you can create an array of different-length strings, whereas even if I have multi-dimensional arrays, byte[][] would have all rows be the same length.

Strings are not null-terminated, they have a length associated with them like arrays do.