Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY



Institute of Computer Science

# Bachelor's diploma thesis

in the field of study Computer Science
and specialisation Computer Systems and Networks

Transpiler for Pax language

Meet Soneji
student record book number 330277

thesis supervisor
dr hab. inż. Ilona Bluemke

Warsaw 2026

**Abstract:** This thesis presents Pax, a statically typed imperative systems-programming language and a transpiler that generates C code. Pax aims to simplify systems programming by reducing syntactic verbosity, removing declaration-order restrictions, avoiding manual header-file management, and supporting compilation of an entire project into one C source file. The language provides name-first declarations with type inference, unambiguous prefix pointer syntax, and length-aware strings and arrays that do not implicitly decay to raw pointers. The Pax frontend performs lexical, syntactic, and semantic analysis before code generation, allowing errors to be reported using the original source code. The transpiler was implemented from scratch as a self-contained executable and uses a custom pool allocator. Its standard library, including string and sorting modules, is written in Pax. The implementation was tested with more than 50 programs comprising over 2,500 lines of Pax code and demonstrated through proof-of-concept applications: a real-time ASCII ray marcher, terminal shaders, and Conway's Game of Life.

**Keywords:** programming language design · transpiler · compiler frontend · semantic analysis · type inference · systems programming · pool allocator · Pratt parser

**Streszczenie:** W pracy przedstawiono Pax — imperatywny język programowania systemowego o statycznym typowaniu — oraz transpiler generujący kod w języku C. Celem języka jest uproszczenie programowania systemowego poprzez ograniczenie rozwlekłej składni, usunięcie ograniczeń wynikających z kolejności deklaracji, rezygnację z ręcznego zarządzania plikami nagłówkowymi oraz kompilowanie całego projektu do jednego pliku źródłowego C. Pax oferuje deklaracje rozpoczynające się od nazwy, inferencję typów, jednoznaczną składnię prefiksową wskaźników oraz łańcuchy znaków i tablice zachowujące informację o długości. Frontend języka wykonuje analizę leksykalną, składniową i semantyczną przed generowaniem kodu, dzięki czemu błędy mogą być raportowane względem oryginalnego kodu źródłowego. Transpiler został zaimplementowany od podstaw jako samodzielny program i wykorzystuje własny alokator pul pamięci. Biblioteka standardowa, w tym moduły łańcuchów znaków i sortowania, została napisana w języku Pax. Implementację przetestowano na ponad 50 programach obejmujących ponad 2500 linii kodu oraz na aplikacjach proof-of-concept: rendererze promieni ASCII, shaderach terminalowych i grze w życie Conwaya.

**Słowa kluczowe:** projektowanie języków programowania · transpiler · frontend kompilatora · analiza semantyczna · inferencja typów · programowanie systemowe · alokator pul pamięci · parser Pratta

# Contents


**1 Introduction** **1**

**2 Intro to Pax language** **3**

**3 Key Features of Pax** **4**

3.1 Syntactic features . . . . . 4

3.1.1 Declaration and type inference using := . . 5

3.1.2 Function Declarations . . . . . 5

3.1.3 Prefix Pointer Syntax . . . . . 7

3.1.4 Array Declaration . . . . . 8

3.1.5 Structs . . . . . 8

3.1.6 Automatic Pointer Dereferencing . . . . . 9

3.2 Semantic features . . . . . 10

3.2.1 Order-Independent Declarations . . . . . 10

3.2.2 String Representation . . . . . 11

3.2.3 Arrays as wide pointers . . . . . 13

3.2.4 Topological Struct Sorting . . . . . 13

3.2.5 Modules and Single Unit Compilation . . . . 14

3.2.6 Builtin Crash Handler . . . . . 16

3.2.7 For Loop . . . . . 17

3.2.8 Defer . . . . . 18

3.2.9 Foreign Functions . . . . . 20

3.3 How does Pax differ from C/C++ . . . . . 20

**4 Transpiler Implementation** **21**

4.1 Transpiler vs Compiler . . . . . 23

4.2 Frontend pipeline overview . . . . . 23

4.3 Lexer . . . . . 24

4.3.1 Token Representation . . . . . 24

4.4 Parser . . . . . 26

4.4.1 Recursive Descent vs Pratt Parser (TDOP) for Expressions . . . 26
4.5 Abstract Syntax Tree Design . . . 29
4.5.1 Why a tree . . . 29
4.5.2 Node structure and inheritance . . . 30
4.5.3 Main AST nodes . . . 31
4.5.4 How the AST changes across phases . . . 34
4.5.5 Overall AST design choices . . . 35
4.6 Semantic Analysis Pipeline . . . 36
4.6.1 Scopes and Symbol Resolution . . . 37
4.6.2 Implementing Order-Independent Declarations 38
4.6.3 Struct and Member Resolution . . . 40
4.6.4 Member Access and Inferred Types . . . 42
4.6.5 Type Inference and Type Checking . . . 43
4.7 C Converter implementation . . . 44
4.7.1 Emitting Debug directive in C . . . 45
4.7.2 Pax vs C, implicit work of C converter . . . 45
4.8 Memory Management with a Pool Allocator . . . 48
4.8.1 Bump Allocation . . . 48
4.8.2 Pool Structure . . . 49
4.8.3 Block List Management . . . 50
4.8.4 Custom Array Implementation . . . 51
4.8.5 Memory Profiling . . . 52
4.8.6 Pool vs Malloc Performance comparison . . . 54
4.9 Error messages . . . 55
4.9.1 Error output format . . . 56
4.9.2 Four phases that throw errors . . . 56
4.9.3 Examplary errors thrown . . . 57
4.9.4 Error recovery . . . 57
4.9.5 Deferred error reporting . . . 58

**5 Proof of concepts** **59**
5.1 Real time ASCII raymarcher . . . 60

5.2 Pluggable shaders . . . . . 63
5.3 Dynamic memory and pointers . . . . . 66
5.4 Conway's Game of Life . . . . . 67
5.5 Built-in Pax modules . . . . . 69
5.6 Negative tests . . . . . 72

**6 Conclusion** **72**

**A Pax Grammar in Backus–Naur Form** **78**

# 1 Introduction

There has been recent focus on developing more modern alternative programming languages to C and C++. This motivation is valid, as having C around for more than fifty years has given us a much better feel for what works and what does not in language design. Their reliability and low level of abstraction still make them the default choice for performance-focused work. At the same time, the C/C++ model has its drawbacks. The most visible of these are the long compile times of modern C++ and its bloated standard library, along with verbose syntax. Meanwhile, header files and forward declarations remain a constant source of manual work.

This project is an attempt to address the above-mentioned issues. This paper is divided into two main parts: the design of a custom programming language called **Pax**, and the development of a complete transpiler frontend pipeline built from scratch. The goal is to show syntactic, semantic and structural choices made for the Pax language and demonstrate the functionality and reliability of both the language and the transpiler with a test suite containing more than 2,500 lines of pure Pax source code.

Rather than building a complete compiler to output machine code directly, Pax takes a transpiler approach and targets C, as outlined in Figure 1. C was selected as the target because it maps very closely to assembly without any "hidden" runtime overhead. Emitting C also means the project can be responsible for everything concerning the language, its parsing, type checking and semantic analysis. The C compiler serves as the backend for now, handling the target-specific code generation and optimization, while the Pax frontend is responsible for everything else. Architecturally, the C converter is a single replaceable stage at the end of the pipeline, and everything before it is written as the frontend of an actual compiler.

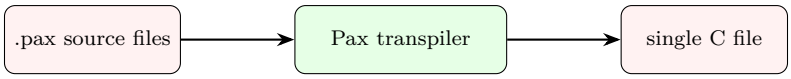



Figure 1: Pax source code is transpiled into a single C file.

It is important to emphasize that Pax is not a translation layer over C. The frontend performs full analysis before any C code is emitted, to find all the compile-time errors. There should never be a need to rely on the C compiler to report an error in the Pax source. This is a real improvement over the usual transpiler approach, where error detection is delegated to the backend compiler. In that setup, errors point to the auto-generated code rather than the user's source, and line numbers lose all meaning after the transformation. In Pax, whenever the user writes invalid code, the error always points to their own source code.

The remainder of this paper is organized as follows. In section 2 the Pax language is introduced: its goals and influences, and how it is different from C and C++. In section 3 the extensions are presented in detail, which can be characterized as syntactic improvements to make programs shorter. Semantic improvements eliminate structural design errors such as declaration order, header boilerplate, and reliance on external build systems.

In section 4 the architecture of the transpiler is described: the frontend pipeline, memory management, and error reporting facilities are all covered.

Using the toolchain, non-trivial programs are illustrated in section 5. These programs combine all kinds of features into long form programs, putting the language to the test. The language's complete BNF grammar is provided in Appendix A.

## 2 Intro to Pax language

Pax is a strongly and statically typed, imperative systems programming language designed to replace the friction of C++ development by reducing verbosity while retaining its performance profile. Pax follows the traditional structured programming paradigm with functions, structs, loops, and conditional statements, but with modern conveniences that eliminate C's syntactic overhead.

Pax's syntax takes inspiration from two main sources to make code easier to read and parse. The core idea of writing variable name first `‘name:type‘` originally comes from Pascal, which proved decades ago that this order is clearer than type-first style used in C. However, Pascal separated declaration from assignment, requiring variables to be declared in a separate block before they can be used. Pax adopts the `‘name:type=value‘` syntax proposed by Sean Barrett [1] and popularized by Jonathan Blow in the design of Jai language [1] [2]. This syntax combines declaration and initialization into a single atomic statement. This allows variables to be initialized immediately at their point of definition while preserving the benefits of the Pascal-style ordering.

Other modern languages have also recognized the above mentioned benefits, though with added syntactic requirements. For example, Go adopts a `‘var name type = value‘` structure [4] for explicit declaration, but requires completely different syntax `‘:=‘` for type inference. Pax gets rid of separation by sticking to the unified model proposed by Sean Barrett, ensuring explicit and inference declarations share the exact same structure.

[1] At the time of writing, Jai language remains in closed beta and has not been publicly released.

### Why Strong Static type checking?

Compile-Time strong typing provides three benefits for systems programming:

1. **Error checking**: Type mismatches, use of uninitialized values, invalid member accesses before program runs is especially important in low-level code where undefined behavior can lead to hard-to-debug memory corruption.

2. **Understandability**: Explicit writing of argument types, return types, and intermediate variables helps us understand the program. When reading unfamiliar code, inspecting its types often can explain its intent more quickly than comments.

3. **Speed**: When types are known at compile time, the compiler can generate efficient code without runtime type checks which slows down performance. Static types enable better inlining, and register allocation, which is crucial since Pax is aims to be as fast as C.

Pax meets these goals through explicit type annotation where it improves clarity and type inference where it removes redundancy.

# 3 Key Features of Pax

The features of the language can be categorized by two parts, it's **syntactic features** which aims to make code more readable and expressive, and **semantic features** which aims to solve structural and architectural issues inherent in C compilation model inspired by modern languages.

## 3.1 Syntactic features

Pax follows standard imperative language syntax with C-like expressions and control flow. The full BNF grammar is available

in section A. Below only the Pax's distinct syntactic features are highlighted.

### 3.1.1 Declaration and type inference using :=

As mentioned previously, in C, every variable requires an explicit type annotation, which can lead to redundancy. Pax has type inference for initialized variables using := operator. This tells the compiler that type is not important for this statement, so rely on the right hand side expression to deduce it's type.

Listing 1: Declaring variables

```
// use : to mean declare, = to mean assign
x := 42;                    // Declare x, implicit type
x: float32;                 // Declare x, explicit type
x: int = 42;                // Declare and initialize with explicit type

student: Student;           // user defined type (struct)
ptr := &student;            // pointer to Student inferred

name := "Timmy";            // string inferred

result := compute(a, b);    // type inferred from functions return value
                            // if no explicit type given
```

As shown in Listing 1, Pax provides different forms of variable declaration to balance convenience and explicitness where needed. Variables declared with ':=' automatically infer their type from the right-hand side expression. This eliminates redundant type annotations while maintaining full type safety. The compiler deduces types during semantic analysis, catching type errors. Explicit type with initializer can be used when implicit type conversion is needed as long as source can be fit into destination variable.

### 3.1.2 Function Declarations

A function's signature follows similar idea of name first as variable declarations. The syntax uses a double colon '::' to bind a function body to an identifier following by lists of parameters and then

return values of the function at the end of the signature rather than beginning like in C.

Listing 2: Function declarations

```
1  // No return type (void)
2  print_msg :: (msg: string) {
3      printf("%s\n", msg);
4  }
5
6  // function that returns type
7  add :: (a: int, b: int) -> int {
8      return a + b;
9  }
10
11 // multiple return types
12 divide :: (a: int, b: int) -> int, int {
13     return a / b, a % b;   // quotient, remainder
14 }
15 q, r := divide(20, 3);
16
17 // a = 0 by default
18 func :: (a: int = 0){
19
20 }
21 func(); // a is passed 0
22
23 // if default parameter is used the parameters following 'a' need to be
      explicitly passed
24 funct :: (a: int = 0, b: int = 0, c: int) {
25
26 }
27 funct(b=4, c=3); // a's default value used, but not b and c. So their
      value must be passed
28
29 ret :: () int, int {
30     a : int = 1;
31     b : int = 2;
32     return a, b;
33 }
34 _ , b := ret(); // first return value not needed, therefore, disregarded
      using '_'
```

Listing 2 demonstrates how different function's signatures are declared. Double-colon '::' syntax distinguishes function declarations from variable declarations. Function parameters also act like normal variable declarations because they can have default values,

thus using ‘:=‘ is allowed. The return type is optional, if not given, the function returns (void). There can be multiple return types and can only get them during function call in the order they are returned. Meaning inorder to get second return value you must get the first return value, but instead of having to give the first value a name even though it is not needed, it can be disregarded by using the ‘_‘ symbol. Moreover, if only the first return value is needed and rest ignored then using ‘_‘ for subsequent return values is not necessary.

### 3.1.3 Prefix Pointer Syntax

To indicate pointer type, caret ‘^‘ is used as a prefix for consistency with type reading direction (left-to-right). Since, the types are seperated by colon in a declaration, placing the pointer ‘^‘ symbol before the type it modifies, the syntax creates clear association between pointer and it’s base type.

Listing 3: Declaring pointer types

```
1 ptr: ^int;      // Pointer to an int
2 a: ^^Student;   // Pointer to a pointer
```

Listing 4: C’s confusing syntax

```
1 // C
2 int* a, b, c; // looks like a,b and c are pointers, but actually b and c
     are of type int
3
4 // In C you must repeat * for each variable
5 int *a, *b, *c;
```

Listing 5: Multi variable pointer declaration in Pax

```
1 // PAX
2 a, b, c : ^int; // all three are pointers!
```

As shown in Listing 3, placing ‘^‘ symbol before the type specifier creates balanced type definition. This solves the common confusing syntax of C shown in Listing 4 where the asterisk binds to the

variable name not the type. So, in Listing 5 the variables `a`, `b` and `c` are all correctly marked as pointers without requiring repetition of pointer symbol.

### 3.1.4 Array Declaration

Array in Pax use square brackets during declaration with the size specified before the type. This syntax clearly distinguishes between arrays, pointers to arrays, and arrays of pointers.

Listing 6: Array declaration

```
1  a: [4]int;   // an array of 4 integers
2  b: ^[]int;   // pointer to an array of integers
3  c: [4]^int;  // array of 4 pointers to int
```

As shown in Listing 6, this syntax keeps the array size and pointer markers visually close to the element type.

### 3.1.5 Structs

In Pax struct syntax is similar to struct in C++ but they do not allow member functions, and unlike classes there is no concept of public and private members whatsoever, hence all members are always public. The struct declaration syntax is also similar to function declaration in Pax. See Listing 7 below.

Listing 7: Struct declaration

```
1  Vec3 :: struct {
2      x: float;
3      y: float;
4      z: float;
5  }
```

There are several ways to initialize a struct such as standard member-wise assignment and designated initializer. They are simply different ways to write the same code.

Listing 8: Struct member assignment

```
1  v1 : Vec3;
2  v1.x = 1.0;
3  v1.y = 2.0;
```

```
4  v1.z = 3.0;
```

Listing 8 is an example of member-wise assignment. Multiple ‘.‘ operators can be used for nested variables.

### 3.1.6 Automatic Pointer Dereferencing

C distinguishes between accessing member of a struct value (`obj.x`) and a pointer (`obj->x`). This adds extra friction when refactoring code, where we may need to change value type to pointer type everywhere in the code. Pax eliminates this by using the same ‘.‘ operator for accessing both value and pointer types.

Listing 9: Pointer auto-dereference

```
1  Student :: struct {
2      name: string;
3      grade: int;
4      supervisor: ^Student;  // Pointer member would be initialized first
5  };
6
7  // Struct member pointer access requires no explicit dereferencing.
8  s: ^Student;
9  s.name = "Timmy"; // Generated C code: ((*s).name = "Timmy");
10 s.grade = 95;     // Generated  C code: ((*s).grade = 95);
11
12 // nested pointers
13 local: Student;
14 s.supervisor = &local; // pointers need to initialized before
15                        //  dereferencing, otherwise its considered
16                        //  a compile time error in Pax
17 s.supervisor.name = "Alan";
18 // Generated C code: ((*(*s).supervisor).name = "Alan");
```

As shown in Listing 9, there is no need for a special syntax to dereference for pointer or value types, because, the same "." notation works. The transpiler automatically inserts dereference operators in the generated C++ code, eliminating the need for special syntax unlike C which has "." and "->" operators. This works recursively, accessing members of pointer-typed struct fields is also dereferenced automatically. The transpiler handles arbitrary pointer depths, simplifying code while maintaining type safety.

## 3.2 Semantic features

Although the syntactic improvements defined in the previous section reduce the visual noise of C developement, syntax alone does not solve the structural friction in the language. Many of the difficulties C/C++ programming do not stem from how the code is written but from the rigorous manual management that is demanded. For example, manually having to update function declaration in the header when it's signature is changed, repeatedly recompiling the same headers in large projects.

Pax addresses these issues by order independent declarations, safer string and array representations, topological sorting of user defined types and module system that supports single unit compilation.

### 3.2.1 Order-Independent Declarations

In standard C and C++, functions and structs must be declared before they are used, hence the neccessesity of header files. This forces developers to manually forward declare and maintain header files, creating extra friction. Moreover, as the program gets larger it starts to become tedious and error prone over time. Pax eliminates this requirement completely.

Listing 10: Functions and Structs order independency

```
1  main :: () {
2      local: Student;   // struct type "Student" used before defined
3      local.name = "Timmy";
4      print_student(&local);  // function "print_student" invoked before
       declared
5  }
6
7  print_student :: (s: ^Student) { // struct type "Student" used before
       defined
8      printf("Name:%s\n", s.name);
9  }
10
11 Student :: struct {   // struct defined at the end
12     name: string;
13     grade: int;
```

```
14 };
```

Functions and structs can be used before their definitions appear in the source file and in any order as shown in Listing 10. This eliminates the need for any forward declarations and header files whatsoever.

### 3.2.2 String Representation

In Pax, a string is a true value with both data pointer and count and is never implicitly shared. It does not evaluate to *char**, unlike in C. The *char** in C is null terminated, with length determined by scanning for terminator character, offering no safety or ownership. In C++ std::string solves this by storing its size however, its known to be heavy weight, relying on heap for storage, hurting performance. Pax solve it as an ordinary struct and not as hidden compiler builtin.

Listing 11: String as a struct

```
1 string :: struct {
2     count: s64;
3     data:  ^u8;
4 }
```

In Listing 11 *count* is the number of bytes in the string, and *data* points at those bytes. Because the length is stored next to the pointer, operations do not need to scan for a `'\0'` terminator to know how long the string is. A string value can also be empty in a simple way: `data = null` and `count = 0`.

Listing 12: Core String.pax operations

```
1 New_String :: (src: string) -> string; // owned copy of a literal
2 concat :: (a: string, b: string) -> string;  // new owned buffer
3 slice :: (s: string, start: s64, count: s64) -> string; // new owned
    buffer
4 equals :: (a: string, b: string) -> bool;  // same count and contents
5 get :: (s: string, index: s64) -> u8;  // bounds checked, 0 out of range
6 destroy_str :: (s: ^string);  // frees the buffer
```

The standard module **`String.pax`** is written in Pax itself. List-

ing 12 shows the most important functions. The functions that return a `string` type allocate a fresh buffer, so a slice never points into the middle of another string, and the caller releases the result with `destroy_str`. `equals` compares both count and contents, and `get` reads a byte with bounds checking. Listing 13 below shows both kinds of strings (heap based and on stack based).

Listing 13: Creating and concatenating strings

```
a := New_String("Hello");  // allocate string on heap
b := New_String(" World"); // another string on heap
c := concat(a, b);         // c.count = 11, c.data = "Hello World"
sub := slice(c, 0, 5);     // sub.count = 5, sub.data = "Hello"

str: string = "Hello";  // on stack
// str.count = 5, str.data = "Hello"

upper := "THIS STRING WAS IN UPPER CASE BUT NOW ITS IN LOWER CASE";
for c: upper { // can also iterator over the string type
    printf("%c", to_lower(c));
}
// output:
// this string was in upper case but now its in lower case

str_len := "testing".count; // use .count directly on the string literal
                            // type inferred (s64)
str_data_ptr :=  "testing".data; // and .data, type inferred (^u8)
```

In Listing 13, show heap strings: `New_String` allocates, `concat` builds a new buffer, and `slice` copies a fresh substring, so every result owns its memory. The rest is readonly: `str` is a static literal, the `for` loop walks `upper` using its `count`, and `"testing".count` and `"testing".data` show that a literal is just a `string` value. This keeps the model simple. A `string` is data that can be passed around like any other struct, ownership is explicit through helpers like `New_String` and `destroy_str`, and because `count` travels with `data`, Pax avoids passing a bare pointer and hoping the length is still known at the use site.

### 3.2.3 Arrays as wide pointers

Unlike in C where arrays automatically cast to pointers, in Pax they are "wide pointers" that contain extra information like the array's size. This prevents common buffer overflow errors and gets rid of the need to pass array sizes as separate parameter.

Listing 14: Arrays as wide pointers

```
a: [4]int;
a[0] = 834;

// Arrays have implicit .data and .count members
printf("a[0] = %d\n", a[0]);           // output: a[0] = 834
printf("a.data = %p\n", a.data);       // pointer to first element
printf("a.count = %d\n", a.count);     // output: a.count = 4

// Pointer to array will preserve size info
b: ^[]int = &a;
b[0] = 999;
printf("b[0] = %d\n", b[0]);           // output: b[0] = 999
printf("b.count = %d\n", b.count);     // still outputs: b.count = 4

// an example
walk_array :: (arr: ^[]int) {
    for i: 0 .. arr.count-1
        printf("%d\n", arr[i]);  // .data and ^[]int are both
                                 // dereferenced here automatically
}
walk_array(&arr); // prints arr contents
```

As seen in Listing 14 the subscript access is uniform, whether a variable is static array `[N]Type` or a pointer to an array `^[]Type`, the syntax `arr[i]` always works without explicit dereferencing.

### 3.2.4 Topological Struct Sorting

While Pax allow order independent declarations in the language, the target C compiler still requires strict ordering of `struct` definitions. The C compiler needs to know the struct's definition and subsequently its size before that type can be used by value in other declarations.

From the programmer's point of view, Pax guarantees that any set of mutually dependent struct can be written in any order, with forward declaration, as long as they are valid. For example, if two structs reference each other through pointers, data structures such as, linked list or tree, it works fine regardless (since size of a pointer is always known). However, if there is a cycle of reference by value, it makes it impossible to determine a well defined order. This is important for the C compiler, because Pax still relies on it for low-level details like struct sizing and layout. So, in that case, Pax will report a compile time error explaining that the structs form circular dependency.

In principle, the language behaves as if all user defined types were automatically sorted into dependency order before any code is generated and any unsortable cycle to be flagged otherwise. This follows C's restriction on value type dependencies, while getting rid of the need for the programmer to manually reason about the order of declaration.

### 3.2.5 Modules and Single Unit Compilation

In C the `#include` preprocessor is a textual replacement operation where files are literally pasted in the source file during preprocessing, before the resulting translation unit is compiled. This design has become major bottleneck for large programs, especially C++ which is known to have extremely slow compilation times as codebase grows.

The conventional model we have adapted to is to split a program into many separate files (translation units), compile each one independently into an object file, and then combine them during the linker phase. In practice, this approach is usually paired with incremental builds and external build systems so that only the changed files need to be recompiled. While this strategy was originally in-

troduced to solve the problem of scaling large programs, it also introduced a new layer of complexity: developers must now manage build scripts, dependency tracking, and build tool syntax in addition to the language itself. As a result, building a C or C++ program often requires learning a separate ecosystem such as Make or CMake, which can feel like solving one problem by creating another.

The outcome is, each translation unit that includes, for example, `<stdio.h>` or another large header ends up pulling in the same long chain of nested includes again and again. The compiler does not remember that it has already parsed those headers for a different source file, so it must re-lex and re-parse the entire header stack for every translation unit. As the number of source files grows, this duplicated front-end work scales almost linearly with the number of translation units. By the time compiler figures out that and starts cleaning up during linking phase, it is already too late.

Therefore, the result is that most of the time, compile times are often dominated not by optimizations or code generation, but by repeatedly re-parsing the same system and project headers. This explains why large C++ projects suffer from very slow builds, despite doing relatively little new work per file beyond handling its small amount of unique code.

Pax replaces this by proccessing the entire project and modules into a single AST by using entry point of the program to recursively process all imported modules within it. And generates a single C file ensuring the C compiler parses the generated code only once, significantly improving build performance.

### 3.2.6 Builtin Crash Handler

While outputting C code, Pax also adds a small crash handler to the generated C file in debug mode. The handler uses the Windows API, and on a runtime crash it walks the call stack and prints a trace with source file names and line numbers, because the generated C contains `#line` directives pointing back to the original Pax source. A stack overflow is handled separately, the handler needs stack space itself, so it can only print `STACK OVERFLOW`.

Listing 15 shows a minimal example. `deref_null_ptr` dereferences a null pointer, which crashes at runtime unless `p = &var;` is uncommented.

Listing 15: Crashing Pax program

```
1  deref_null_ptr :: () {
2      p: ^s64 = null;
3      var := 0;
4      // p = &var; // uncommenting this line prevents the crash
5      *p = 42;
6  }
7
8  main :: () {
9      deref_null_ptr();
10 }
```

The handler is prepended to the generated C file and installed at the start of `main`. Running the program produces Listing 16: the first frame is the exact crash line inside `deref_null_ptr`, and the second is the call in `main`.

Listing 16: Runtime crash output in console

```
1   --- RUNTIME CRASH STACK TRACE ---
2
3  ******* C:\pax\tests\demo_runtime_crash_handler.pax:5  deref_null_ptr
4          C:\pax\tests\demo_runtime_crash_handler.pax:10  main
5
6   --- END OF STACK TRACE ---
```

A stack overflow is different, shown in Listing 17: `infinite_recurse` fills the stack, and since the handler needs stack to run, the only output is `STACK OVERFLOW`, as in Listing 18.

Listing 17: Infinite recursion

```
infinite_recurse :: () {
    infinite_recurse();
}
```

Listing 18: Stack overflow output in console

```
  STACK OVERFLOW
```

### 3.2.7 For Loop

C and C++ already provide a powerful and flexible `for` loop. That flexiblility is useful but for the most common cases, such as counting over a fixed range or walking an array, it forces programmer to write down the initialization, termination and updator by hand. Pax takes the syntax of variable declaration (see 3.1.1) and the same idea of a controlled loop but offers more declarative syntax for these common cases so its clearer without replacing the power of a general loop construct.

Listing 19: For loops

```
// range: start inclusive .. end inclusive
for i: 0..4 {
    printf("i=%d\n", i); // i takes values 0, 1, 2, 3, 4
}

arr: [5]int;  // array of integers of size 5

// index based iteration using the array's .count
// arr.count-1 is the last valid index
for (i: 0 .. arr.count-1) {
    printf("arr[i]=%d\n", arr[i]);
}

for a: arr {  // element based iteration
    printf("%d ", a);   // a is each element of arr
}
```

As shown in Listing 19, Pax provides two main forms of `for` loops. The counted form `for(name:start..end)` iterates a loop variable over a range where both `start`and `end` are included. In the first example, `for i:0..4` therefore visits the values `0`, `1`, `2`, `3` and `4`. The loop variable is local to the loop body, and the bounds may be either constant literals or runtime expressions.

When indexing an array is required, the same counted form combines with the array's implicit `.count` field. The second loop in Listing 19 uses `for(i:0..arr.count-1)` to walk every valid index using the array's implicit count as a runtime expression. If the array length later changes, the loop will still cover the full range of elements.

The third form, `for a:arr`, iterates directly over the elements of an array. Here the loop variable `a` is bound to each element, so no explicit index is needed. This range based style is preferred when only the element values matter and the position inside the array does not.

Together, these forms keep iteration concise while remaining fully typed and bounds aware.

### 3.2.8 Defer

Pax provides a `defer` statement that schedules code to run when the current scope exits. This is useful for cleanup that should always happen, even if the function returns early or an error path is taken: closing a file, unlocking a mutex, or printing final diagnostics. Without such a construct, cleanup must be duplicated before every `return`, which is easy to miss when the control flow grows.

Listing 20: Defer statements

```
1
2  printf("1\n");
```

```
3  defer printf("5\n");  // defer keyword before a statement
4  printf("2\n");
5  defer printf("4\n");  // defer keyword before a statement
6  printf("3\n");
7
8  // Outputs (LIFO order):
9  //  1
10 //  2
11 //  3
12 //  4
13 //  5
```

Listing 21: Defer statement (practical version)

```
1
2  ptr := malloc(sizeof(int)*10);   // allocate memory on heap
3  defer {    // you can defer entire block of code too
4      printf("Freeing memory...\n");
5      free(ptr);                // put a defered free on that allocated mem
6                                // early
7  }
8  // Do some work on the malloc'd memory
9  printf("End work \n");
10
11 // OUTPUT:
12 //   End work
13 //   Freeing memory...
```

As shown in Listing 20, deferred statements run in reverse order of registration (last-in, first-out) when the enclosing scope ends. The first example therefore prints `1`, `2`, `3`, then the deferred `4`, then `5`. A `defer` may take either a single statement or a full block. In Listing 21, a block is deferred after `malloc`: normal work runs first (`End work`), and only when the scope exits does the deferred block run, printing `Freeing memory...` and calling `free`. Keeping allocation and release next to each other makes the cleanup hard to forget, even if later returns or early exits are added above the end of the function.

C has no built-in equivalent, so, cleanup must be written by hand before each exit path, or factored into helper functions that the programmer must remember to call.

C++ solves it with RAII (Resource Acquisition Is Initialization), a local object's destructor runs automatically when the object goes out of scope, so cleanup is tied to object lifetime rather than to an explicit statement.

Pax does not use RAII so, `defer` is introduced as an explicit way to schedule cleanup when current scope ends.

### 3.2.9 Foreign Functions

Pax can call C functions directly. A foreign declaration tells the Pax compiler that the function body already exists, the Pax program usually in the C library or in linked C object code, so Pax should trust that the symbol will be available at link time. Foreign function signatures are declared by the **`#foreign`** directive at the end of a function signature in Pax language. See Listing 22:

Listing 22: Foreign function signature

```
1 malloc :: (size: s64) -> ^void #foreign;
2 free   :: (ptr: ^void) #foreign;
3 printf :: (fmt: ^u8, ..) -> int #foreign;
```

The frontend still type checks calls using the declared signature, but it does not expect or generate a body. In the emitted C, the call is simply outputted to C. This is how Pax code uses routines like as `malloc`, `free`, and `printf` without reimplementing them, since that is not the goal of this project.

## 3.3 How does Pax differ from C/C++

Pax is intentionally close to C and C++. It remains a statically typed systems language. The differences introduced in this chapter is not the discourage these languages but a set of adjustments aimed at reducing everyday friction.

At the syntactic level, Pax prefers name-first declarations. For ex-

ample, a function declaration, after writing function name the operator ‘::‘ follows and this instanly makes it easier to find function declaration in a large code base, because we can regex search with the function name followed by ‘::‘ operator and we are sure its the function declaration. In C/C++ this is not the case because its syntax does not make it easy. Atleast in C++ there is scope resolution operator which has the syntax `classname::function_name`.

Pax also has optional inference see (3.1.1), a single `"."` operator for both member accesses by value and by pointer (Listing 9), and more declarative forms for common loops and array iteration (Listing 19). At the semantic level, it removes much of the manual bookkeeping associated with headers and forward declarations through order-independent definitions, modules, and single-unit compilation (Listing 10). Arrays and strings also carry their own size information, rather than decaying into bare pointers as in C.

These choices are not meant to replace the power of C or C++. C style control flow, explicit types where they help understanding, and low level control stays in focus. What Pax really aims to improve is the that we can have all of the good things from C and C++ with less redundant annotation, fewer declaration-order constraints, and less dependence on the preprocessor and external build machinery for basic modularity.

With the language design now outlined, the next chapter moves from design to practice, presenting the architecture and implementation of the Pax transpiler itself.

# 4 Transpiler Implementation

The Pax transpiler has been implemented completely from scratch, with no reliance on any external lexer generators, parsing libraries

or compiler framework. Every single component from the lexer and top down recursive decent parser to AST generator and C code outputter was hand written specifically for this project. The implementation itself is written in very simple subset of C++. The goal was to keep codebase lightweight and very fast to compile. The C++ standard library is avoided to reduce compilation, since its header based, template heavy design can significantly increase compile and link times when those headers are included [3]. C++ is only used for templates, inheritance, constructors, destructors, member functions on structs. Plain C has none of these. The rest of code stays close to C style.

This approach was chosen for several reasons. First, it provides complete control over error messages, allowing the transpiler to generate precise, context-aware diagnostics that is tailored to Pax's semantics rather than generic parse errors.

Second, building from first principle allows deep understanding of each compilation stage, making it easier to implement Pax specific features, without combating the generic assumptions of the third party tool.

Third, avoiding third party dependencies on parser generators like Yacc/Bison or ANTLR eliminates external complexity and build time overhead. So, by keeping the transpiler self contained, the entire codebase remains testable, debuggable, and maintainable. The transpiler itself compiles quickly (less than a second), and the resulting executable has no runtime dependencies making it easily portable without fighting with library versioning or any instalation requirements.

Finally, this approach serves as the pedagogical goal of this paper, demonstrating that modern compiler construction is possible with-

out any external toolings, as well as the added overheads that come with it.

## 4.1 Transpiler vs Compiler

A transpiler in the traditional sense converts high level source language to a destination high level langauge, then relies on the destination language's compiler to generate machine code. A compiler by contrast can directly translate high-level code into lower-level representations such as assembly, object code, or machine code, without requiring another language's toolchain.

However, calling Pax just a "transpiler" is underselling what it actually does. Pax is a full compiler that does complete lexing, parsing, semantic checking, and type inference before generating any C code. Unlike naive source-to-source translators, Pax catches errors such as, undeclared variables, type mismatches, undeclared function calls and many more during lexing, parsing and analysis stages, before the C compiler ever sees the code. The C compiler is purely used for backend code generation and optimization, not for any error checking. The idea is if your Pax code compiles, the generated C code is guaranteed to be correct, with no errors from C compiler, otherwise, its a bug in the Pax transpiler.

This design allows the project to greatly focus on language design rather than low level code generation.

## 4.2 Frontend pipeline overview

Figure 2 gives an abstract overview of the Pax frontend pipeline, from a `.pax` source file to the generated C translation unit. Starting from Pax source code, the lexer and parser run in parallel to produce an abstract syntax tree (AST) that captures the syntactic structure of the program. The semantic analyser then traverses this tree,

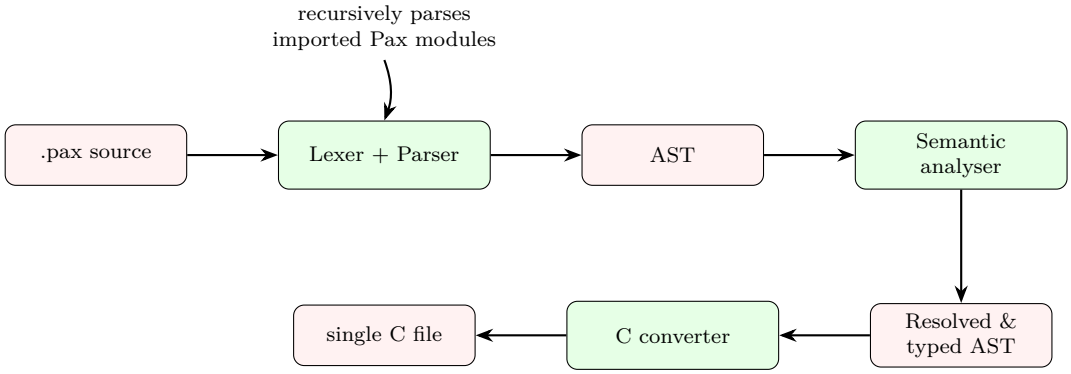



Figure 2: Pax frontend phases (green) and their intermediate results (red).

resolving symbols, doing type inference, and attaching type info so that the program becomes a resolved and typed AST. Finally, the C converter takes this semantically checked representation and emits a single C file, preserving original Pax program's behavior, while making implicit behaviors in Pax explicit in the generated code.

## 4.3 Lexer

The lexer is the first phase of the Pax compiler and also the simplest one. It reads the source file character by character and groups the input into tokens. Pax uses a handwritten lexer rather than a generated one. This gives us complete control over error reporting, source locations, and Pax specific syntax such as `::`, `..`, and `->`.

The lexer keeps track of the current line and column while scanning. Every token stores its original source position so later phases can report errors against the programmer's Pax source, not against generated C.

### 4.3.1 Token Representation

Tokens are defined in `token.h`. The representation is deliberately lightweight is provided in Listing 23:

Listing 23: Token representation

```
1  struct Token {
2      TokenType type;
```

```
3       int row;
4       int col;
5
6       union {
7           const char *value;
8           unsigned long long int_value;
9           double float64_value;
10
11          struct {
12              unsigned long long count;
13              unsigned char *data;
14          } string_value;
15      };
16  };
```

A few design decisions matter here. A `union` is used so different literal kinds do not waste space. Identifiers keep a pointer into the original text rather than always copying the buffer. `TokenType` has explicit entries for Pax keywords and multi-character operators, and string tokens store both `data` and `count`, matching Pax's "sized string" model instead of bare C `char *`.

The lexer itself holds the full source buffer, a current position, live cursor coordinates, and a pointer to the pool allocator used for token storage. `nextToken()` skips whitespace and comments, remembers the location, then classifies the next character into punctuation, operators, numbers, strings, or identifiers. Keywords are recognised inside identifier scanning. `peekNextToken()` can look ahead without permanently moving the cursor, which the parser uses when it needs one token of lookahead.

One small but important detail: when scanning numbers, a single `"."` may start a float, but `".."` must stay a range token. So `"1.."` is not treated as a malformed float; it remains valid for `for i:0..n` style ranges.

## 4.4 Parser

The parser consumes the token stream and builds the Abstract Syntax Tree. Like the lexer, it is completely handwritten. It is responsible for recognising declarations, statements, and expressions, and for attaching source locations to every AST node.

Statements and declarations are parsed with recursive descent. Expressions originally used recursive descent as well, but that version was later replaced by a Pratt parser [5].

### 4.4.1 Recursive Descent vs Pratt Parser (TDOP) for Expressions

The old expression parser was classical recursive descent. Each precedence level had its own function. A simplified version of that idea is demonstrated in Listing 24.

Listing 24: Recursive-descent expression parsing (old, simplified)

```
parseExpression():
    left := parseAdditive()
    while current is comparison or DOT:
        op := current
        advance()
        right := parseAdditive()
        left := Binary(left, op, right)
    return left

parseAdditive():
    left := parseTerm()
    while current is + or -:
        op := current
        advance()
        right := parseTerm()
        left := Binary(left, op, right)
    return left

parseTerm():          // * and /
    left := parseFactor()
    while current is * or /:
        ...
    return left

```

```
parseFactor():        // literals, idents, calls, (expr), unaries
    if unary operator:
        return Unary(...)
    if number/float/string:
        return Literal(...)
    if identifier:
        if next is '(': return parseCall()
        return Ident(...)
    if '(':
        expr := parseExpression()
        expect ')'
        return expr
```

This style is easy to follow. One function maps to one precedence level, so stepping through `a + b * c` is straightforward: multiplication is handled deeper in the call chain than addition. The downside is the amount of repeated structure. Every level has almost the same `while` loop, and adding a new operator means finding the right function and wiring it in by hand. The call stack also gets deeper as more precedence levels are added and this especially becomes painful to debug.

The current expression parser is a Pratt / top-down operator-precedence parser. Instead of one function per precedence level, there is one recursive loop driven by a precedence table as shown in Listing 25:

Listing 25: Pratt expression parsing (current, simplified)

```
parseExpression(minPrecedence = 0):
    left := parsePrimaryOrUnary()
        // literals, idents, calls, parentheses, unaries

    while true:
        prec := getPrecedence(current)
        if prec < minPrecedence:
            return left

        // special cases such as arr[i] can be handled here

        op := current
        advance()
        right := parseExpression(prec + 1)   // left-associative
```

```
15          left := Binary(left, op, right)
16
17      return left
18
19  getPrecedence(type):
20      DOT, [          -> 100
21      *, /            -> 90
22      +, -            -> 80
23      ==, !=, <, ...  -> 70
24      &&              -> 60
25      ||              -> 50
26      otherwise       -> -1
```

Here precedence lives in one place. And `parseExpression()` keeps folding operators as long as the next operator is strong enough compared with `minPrecedence`. Calling `parseExpression(prec + 1)` on the right-hand side makes operators left-associative without extra flags. Unary operators are handled in the primary/unary step with a high fixed precedence.

For Pax the Pratt parser approach is a better fit. Adding or reordering operators is mostly a table change, not a rewrite of several recursive-descent helpers. The expression code is also shorter, which makes it easier to maintain next to all of the statement parsing that still uses normal recursive descent. Statement-level code simply calls `parseExpression()` with default precedence 0.

The old recursive-descent expression parser remains commented in the source as a comparison point. It works, and it is very readable, but for expression-heavy input the Pratt version is more compact and easier to extend. The rest of the parser did not need the same change: `declarations`, `blocks`, `if`, `while`, `for`, and `statements` stay recursive descent because those constructs are irregular and already map cleanly to one function each.

## 4.5 Abstract Syntax Tree Design

Once the parser finishes, the program is no longer text and no longer a flat list of tokens. It is an Abstract Syntax Tree (AST). From this point on, almost every later phase of the compiler works on that tree: symbol resolution walks it, type inference annotates it, and the C converter eventually prints code from it. In that sense the AST is the real intermediate representation of Pax, not just a temporary parse result.

### 4.5.1 Why a tree

A compiler needs some structured form of the program after parsing. A raw token list is too flat because its basically text with punctuations removed. Without a proper structure the relationships between parts of the program are difficult to retain. "Abstract" here means that some concrete source details are dropped once they are no longer needed.

A tree is a natural fit for programming languages. Nested syntax becomes nested structure. A function body is a block node whose children are statements within that function. An `if` statement has a condition expression and one or two branch blocks. A binary expression has a left child, an operator, and a right child. Parentheses, braces, and precedence rules from the source are turned into parent/child links in the tree.

This structure matters for later phases. For example, type checking an expression means checking its children first, and then combining their types. Whereas, resolving a name means searching scopes outward from the current block. All of those operations are generally recursive tree walks. A tree just matches both the grammar of the language and the way the later phases actually work and operate on it.

### 4.5.2 Node structure and inheritance

The AST is implemented with a shallow C++ struct hierarchy. There is a common root type that every node derives from, either directly or through a small intermediate base, like expression or statement. This shared root is able carry common information to the whole tree. A tag saying what type of node it is, and the exact source location where it maps in the source file.

In practice this means two layers of information. The root gives every node a type tag and file/line/column data. Intermediate bases then add fields shared by a whole family of nodes. For example, all expressions share an `inferred_type` field that starts empty after parsing and is filled during semantic analysis. Concrete nodes finally add only the fields they need.

Listing 26: Shared AST bases (simplified from `ast.h`)

```
struct Ast {
    Ast_Type type;
    int line_number;
    int character_number;
    const char *file_name;
};

struct Ast_Expression : public Ast {
    Ast_Type_Definition *inferred_type;  // filled later
};

struct Ast_Statement : public Ast {
    Ast_Statement(Pool* = nullptr) { type = AST_STATEMENT; }
    Ast_Type_Definition *type_definition = nullptr;
    Ast_Expression *expression = nullptr;
    Ast_Block *block = nullptr;
    bool is_return = false;
};
```

This is useful for two reasons. First, later phases can walk “any AST node” or “any expression” without caring about the subtype until they need it, and then cast back with `static_cast<>` when

specific node kind is required. Second, source locations are part of every node from the moment it is created. Nodes themselves are allocated from the custom pool allocator, so the whole tree has a common shared memory pool with same lifetime and can be freed together when compilation ends.

### 4.5.3 Main AST nodes

Below the main families of AST nodes are highlighted, such as expression, statements, declarations, types and so on.

#### Expressions

Expression nodes represent values and computations. They all share the expression base, so each one can later carry an inferred type. The important expression nodes are outlined in Listing 27.

Listing 27: Selected expression nodes (simplified)

```
struct Ast_Literal : public Ast_Expression {
    Value_Type value_type;     // number, float, string, bool, null
    int64_t integer_value;
    double float_value;
    const char *string_value;
};

struct Ast_Ident : public Ast_Expression {
    const char *name;
};

struct Ast_Binary : public Ast_Expression {
    Binary_Op op;              // +, ==, =, ., [], ...
    Ast_Expression *lhs;
    Ast_Expression *rhs;
};

struct Ast_Unary : public Ast_Expression {
    Ast_Unary_Op op;           // - ! & *
    Ast_Expression *operand;
};

struct Ast_Procedure_Call_Expression : public Ast_Expression {
    Ast_Ident *function;
    Ast_Comma_Separated_Args *arguments;
};
```

`Ast_Literal` nodes hold constant values together with a tag describing whether they are integers, floats, strings, booleans, or `null`. `Ast_Ident` stores a name; immediately after parsing it is just text, and later resolution links it to a declaration. `Ast_Binary` represents an operator with left and right children, and is reused for arithmetic, comparison, assignment, member access (`.`), and array indexing. `Ast_Unary` wraps a single operand with operators such as negate, not, address-of, or dereference. `Ast_Procedure_Call_Expression` records the callee and its argument list, which is itself stored in an `Ast_Comma_Separated_Args` node that is shared by calls, multi-assignment, and multiple return values. Finally, `Ast_Cast` and `Ast_Named_Argument` cover explicit casts and named arguments.

So, for example, an expression like **`add(a, b + 1)`** becomes a call node whose argument list contains an identifier and a binary add node, and that binary node itself has an identifier child and a literal child. The hierarchy of the source expression becomes a hierarchy of expression nodes.

### Statements and control flow

Statement nodes represent sequencing and control rather than pure values, Listing 28 shows this.

Listing 28: Selected statement nodes (simplified)

```
1  struct Ast_Block : public Ast_Statement {
2      Ast_Block *parent;
3      Array<Ast_Statement *> statements;
4      bool is_global_scope;
5      bool is_scoped_block;
6  };
7
8  struct Ast_If : public Ast_Statement {
9      Ast_Expression *condition;
10     Ast_Block *then_block;
11     Ast_Statement *else_block;
12 };
13
14 struct Ast_For : public Ast_Statement {
```

```
15      Ast_Ident *variable;
16      Ast_Expression *start;     // range form
17      Ast_Expression *end;
18      Ast_Expression *array;     // for x : arr
19      Ast_Block *block;
20  };
21
22  struct Ast_Defer : public Ast_Statement {
23      Ast_Block *block;
24  };
```

`Ast_Block` represents a sequence of statements and also serves as a scope. `Ast_If` and `Ast_While` carry condition expressions and the corresponding body blocks. `Ast_For` covers both range-based and array-based loops by storing either a `start`/`end` pair or a single `array` expression, and `Ast_Defer` keeps the block that should run when the surrounding scope exits. A small `Ast_Break` node stands for loop exits. Together, these nodes capture the control flow of a Pax program in a form that later passes can work with directly, whether they are resolving names, checking types, or lowering loops and branches into C.

### Declarations

`Ast_Declaration` is one of the central node types in the whole compiler. Instead of inventing separate nodes for every kind of named thing, Pax reuses one declaration node for variables, function parameters, struct fields, and function definitions. A declaration can therefore carry an identifier, optional declared type, initializer, parameter list, return types, and flags such as `is_function`, `is_foreign`, `initialized`, and `inferred`. Multiple return types are stored as an array, matching Pax multi-return functions. Later phases attach even more meaning to this same node rather than replacing it.

#### Types, structs, and modules

Type structure is also part of the tree. `Ast_Type_Definition` covers general types, including pointer targets through `pointed_to_type`. `Ast_Array_Type` stores element type, size expression, and whether the array is resizable. `Ast_Struct` represents named structs with member declarations, and `Ast_Import` stores module import paths for whole-program processing. Builtin types are also represented as type-definition nodes so that inference and checking can treat user types and builtin types uniformly.

### 4.5.4 How the AST changes across phases

A useful way to think about the AST in Pax is that its *shape* is mostly fixed after parsing, but its *information content* grows over time.

#### After parsing

At this point the tree is mostly syntactic. The compiler knows that something is a binary add, a declaration, a call, or a for loop, and it knows where each construct came from in the source file. What it does not yet fully know is what every name refers to, or what every expression's final type is. Some type nodes may still be unresolved placeholders. Declarations may not yet be marked initialised or inferred.

#### During symbol resolution

Resolution walks the tree and connects uses to definitions. Identifiers are matched to declarations through scope chains rooted in blocks. Function calls are linked to function declarations. Member accesses are checked against struct definitions. References that cannot be resolved on the first pass can be queued and retried, which is how order-independent declarations are supported. After this phase, the tree still has the same general shape, but many nodes

now “know” what they refer to.

#### During type inference and checking

This is where expressions gain `inferred_type`. Literals get default types, operators impose constraints on their operands, calls check argument and return types, and explicit casts are validated. Declarations may receive inferred types from their initializers. Auto-dereference for member access is also reasoned about here. Again the tree is not rebuilt from scratch: existing nodes are annotated, and invalid programs are rejected with source-accurate errors.

#### During C generation

By the time the converter runs, the AST is a fully checked program representation. The backend walks it and emits C text: declarations become C declarations, blocks become compound statements, for loops become C loops, defer nodes become code placed at scope exits, and so on. Source locations on the nodes are also used to emit `#line` information so debugging can still refer to Pax source.

### 4.5.5 Overall AST design choices

A few concrete choices follow from the above. Designing the AST is one of the more difficult parts of the frontend, because almost every later phase depends on getting it right, otherwise they get messy quickly. Also, during the development period as the language evolves and more features are introduced the AST nodes naturally evolve as well. The tree is also intentionally mutable across phases. Rather than lowering into a separate representation after every pass, Pax keeps using the same AST and gradually makes neccessary and concrete changes, until there is time for emitting C. This is important since, one of the main goals of the frontend is to report proper errors and debug information in terms of the original program, source locations are treated as part of that core

structure, and every important node points back to the right part of Pax source code for error reporting.

## 4.6 Semantic Analysis Pipeline

After parsing, the compiler has an AST that represents the structure of the Pax program. However, the tree does not yet contain enough information to generate valid C code. Semantic analysis is the stage that gives the parsed program its meaning and makes sure its valid. It checks if names exist, variables are used correctly, functions are called with valid arguments, struct members exist, and expressions use correct compatible types. It also does type inference for declarations that use `":="` operator. The result is the same AST as before, but now its nodes contain the symbol and type information required by the code generator.

This is by far the most complex part of Pax frontend. Lexing and parsing are usually the easiest and simplest parts to implement because they mostly process the source from left to right. Semantic analysis on the other hand cannot always do that, because Pax is designed for declarations to be used before they appear in the source file as mentioned in 3.2.1. A function can be called before even seeing definition, a struct type can be used before its defined. In a long program this can get complicated very quickly since there may be many chained dependencies that turn out to not be resolvable at the first pass and need to be deferred for second pass to attempt resolve again.

The semantic analyser itself has two stages:
**Symbol Resolution:** This stage resolves symbols and validates the structure of the program. It walks the AST, keeping track of nested scopes, declares declarations into scope stack, and tries to find and match every identifier to the declaration that introduced

it. The resolve phase has two passed since Pax is designed as a multipass frontend. After the first pass the whole program has been visited, second pass tries again, because at that point declarations that appeared later in the file or in imported modules are already known to the compile .

**Type Checking and Inference:** Once all names and user defined types (structs) are known, the compiler does type inference and checks types for expressions, declarations, assignments, calls, and return statements.

### 4.6.1 Scopes and Symbol Resolution

Pax uses ordinary lexical scoping. A declaration is visible in the block where it is introduced and in nested blocks below it. When the compiler sees a name, it searches the current block first and then the parent scopes until it finds a matching declaration.

Listing 29: A local declaration shadows an outer declaration

```
1
2  value := 10;  // outer
3
4  main :: () {
5      value := 20; // inner
6      printf("%d", value); // prints 20
7  }
```

In Listing 29 the inner `value` is used inside `main` because the closest matching declaration is selected. Declaring the same name twice in one scope is an error, but using the same name in a nested scope is allowed. This keeps local names independent while making lookup predictable.

During the first traversal, it keeps a stack of active blocks. When entering a function body, loop body, or a scoped block, it pushes that block onto the stack and when leaving the block pops it. Figure 3 shows this first pass. The scope stack is only temporary traversal

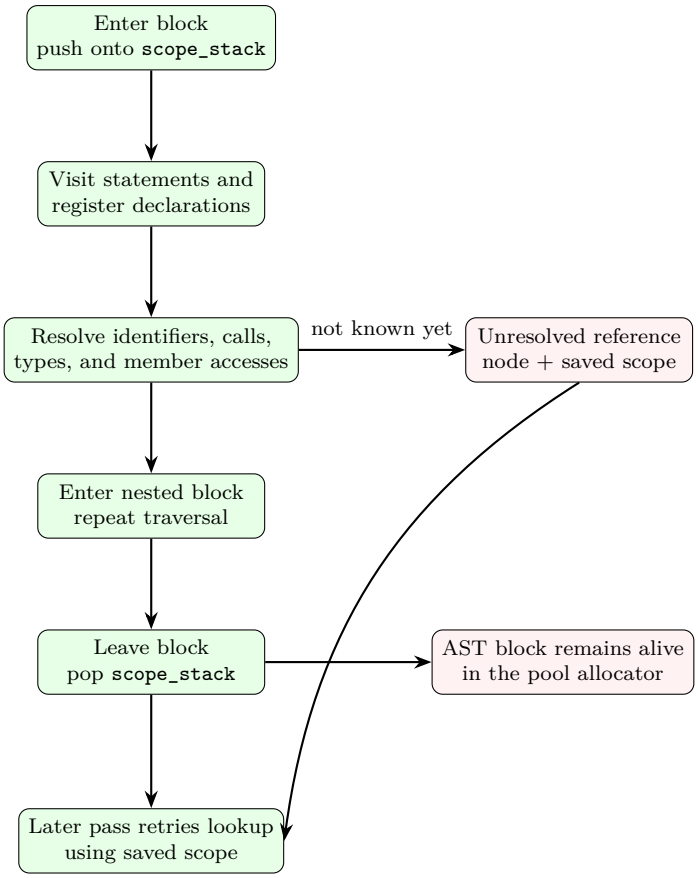



Figure 3: First-pass symbol resolution in Pax. The active scope stack follows the current traversal, while unresolved references store their original AST block so that later deferred passes can perform lookup in the same lexical context.

state: popping a block does not destroy it or the declarations inside it. The whole AST is allocated from the compiler's pool and stays alive for the entire compilation. Unresolved references store the AST block they appeared in together with the saved scope, so a later pass can retry the lookup in the same lexical context without traversing the whole AST again. This matters because a deferred lookup must use the same lexical context as the original reference. Even after a block has been popped from the stack, it still exists, so a later pass can search outward through its parent scopes exactly like the first pass did.

### 4.6.2 Implementing Order-Independent Declarations

Order-independent declarations were introduced earlier as one of Pax's semantic language features, see Section 3.2.1. From the pro-

grammer's point of view, this means that functions and structs may be used before their definitions appear in the source file. The important question for the implementation is how the frontend can support this behaviour and still report actual missing identifiers as errors.

This is why, a single top-to-bottom semantic pass would not be enough. When the compiler encounters a call or type name, its declaration may appear later in the same file or even in a module that has already been parsed but has not yet been visited during the current traversal. Instead of treating this immediately as an error, the first resolution pass resolves every name that is currently visible and records the rest of the remaining references for second pass.

Listing 30: A function call resolved after the first traversal

```
1 main :: {
2     result := double(21);
3     printf("%d", result);
4 }
5
6 double :: (value int) -> int {
7     return value * 2;
8 }
```

As we can see from Listing 30 when the first pass reaches `double(21)`, the declaration is not seen. The compiler stores the call together with the scope in which it appeared and continues traversing the AST. After the first pass has visited all declarations, a later resolution pass repeats the lookup. The call can then be connected to `double` function, and its arguments and return type can be checked normally.

The same mechanism is used for unresolved variable references, user defined types (structs), array related declarations, and member accesses. Each unresolved entry keeps both the AST node and its

original lexical scope. Keeping the scope is necessary because a later lookup must behave as though it were performed at the original source location, rather than from whichever block happens to be active when the later pass runs.

Only after both the first and second pass were not able to find the required definition we are sure that it does not exist in the program and can go ahead and report the appropriate error.

### 4.6.3 Struct and Member Resolution

A member access is represented in the AST as a binary `"."` expression. The lhs is the value whose member is being accessed, while the rhs is the member ident. For a simple expression, like, `player.name`, the semantic analyser resolves `player`, determines its struct type, and searches that struct for a field `"name"`.

Nested accesses require the same operation to be applied repeatedly from left to right. The following example in Listing 31 is intentionally small, but it demonstrates both recursive member resolution and Pax's automatic pointer dereferencing.

Listing 31: Nested member access with an intermediate pointer

```
1  Stats :: struct {
2      score int;
3  }
4
5  Player :: struct {
6      stats ^Stats;
7  }
8
9  main :: {
10     player: Player;
11     s_local: Stats;
12     player.states = &s_local;
13
14     player.stats.score = 100;  // have a look at this line
15 }
```

The parser represents the left-hand side of the assignment as the

following left-associated expression:

(`player.stats`).`score`

The resolver cannot search for **`score`** immediately. It must first resolve **`player.stats`**, because the type of that expression determines the struct whose members may contain **`score`**.

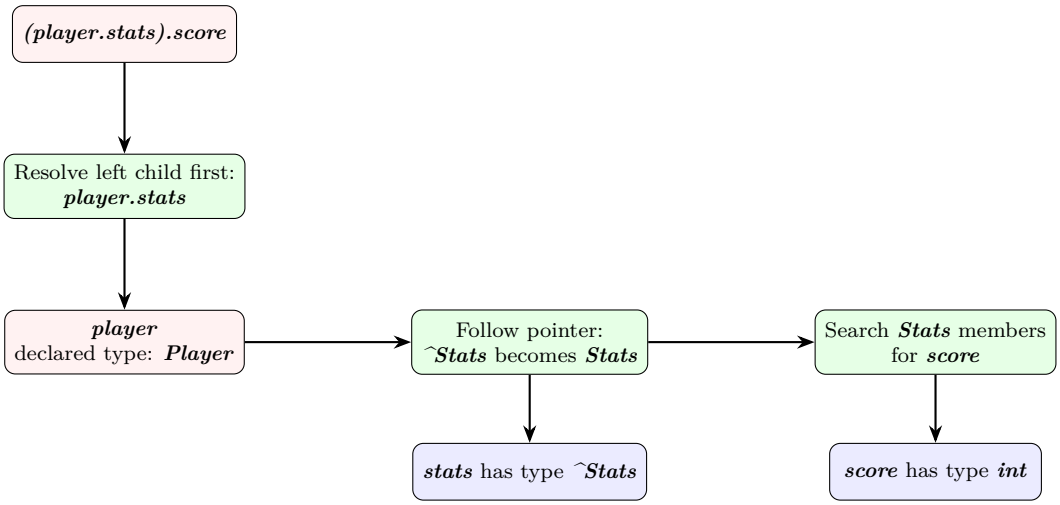



Figure 4: Member resolution for an examplary expression: `player.stats.score`.

Figure 4 above shows the order used by `resolveMemberAccess()`. The outer dot node first recursively resolves its left child, (***player.stats***). The identifier ***player*** is looked up in the current lexical scope and has type ***Player***. The resolver then searches the members of ***Player*** and finds ***stats***, whose declared type is ***^Stats***.

Before resolving the next member, the resolver follows the pointer and uses ***Stats*** as the base struct type. It then searches the members of ***Stats*** for ***score***. The matching field has type ***int***, so the complete expression ***player.stats.score*** is assigned the type ***int***.

This pointer step is the semantic basis of Pax's automatic dereferencing. The programmer writes ***player.stats.score*** whether ***stats*** is stored as a struct value or as a pointer to a struct. The semantic analyser records the resolved field and its type, the C converter later emits the explicit pointer dereference required by the generated C

code.

If the left hand expression has no available type during the first traversal, for example because it depends on a struct declared later in the file, the dot node is placed in unresolved member access queue together with its original scope so that a later resolution pass can retry the same lookup. If the base resolves to a non-struct type, or if the required member does not exist, the compiler reports a semantic error.

### 4.6.4 Member Access and Inferred Types

Most member accesses can be resolved immediately once the type of the expression on the left of `"."` is known. There is one important exception. Sometimes the lhs has no explicit type because it was declared using inference, or because it is itself another member access expression.

Listing 32: A member access whose base type is inferred

```
1  Config :: struct {
2      database: Database;
3  }
4
5  Database :: struct {
6      host: string;
7  }
8
9  main :: {
10     config := make_config();  // some function
11     local_host := config.database.host; // access "host"
12     printf("%s\n", local_host);
13 }
```

To resolve `config.database.host` on line 11 in Listing 32, the compiler must first know the type of `config`. In this example, that type comes from the return type of `make_config`. It can then resolve `config.database`, whose type is `Database`, before finally resolving `host` as a `string`.

For this reason, member resolution can perform a small amount of type inference when the required type has not already been attached to the base expression. This does not replace the later type-inference pass. It is only an exception needed to find the struct definition required for member lookup itself.

### 4.6.5 Type Inference and Type Checking

After name resolution phase is successful, Pax moves to type inference. Every expression receives a type that is stored in its AST node. Literals have default types: integer literals are `s64`, floating-point literals are `float`, string literals are `string`, and boolean literals are `bool`. The type of a variable declared with `:=` is taken from its initializer, see below in Listing 33.

Listing 33: Examples of inferred types

```
1  count := 42;          // s64 (integers are s64 by default in Pax)
2  ratio := 3.5;         // float32
3  name := "Pax";        // string struct
4  enabled := true;      // boolean
```

For an explicitly typed declaration shown in Listing 34, the compiler checks that the initializer can be represented by the requested type. This is especially useful for integer constants, where a value can be numeric but still be too large for the selected integer type.

Listing 34: Checking an explicit declared type

```
1  small : s8 = 100;        // valid
2  too_large : s8 = 1000;   // error
```

Type information then flows through expressions. Arithmetic operators check their operands and produce a result type, comparisons produce `bool`, a function call receives its declared return type, and a member expression receives the type of the field it accesses. Assignments and return statements are checked against the type expected by their destination or enclosing function.

## 4.7 C Converter implementation

The Pax frontend is a hand written pipeline, shown in Figure 2. The lexer and parser turn the source code into an abstract syntax tree, the semantic analyser resolves symbols, checks types and attaches type information, and then the C converter walks the fully resolved tree and emits a single C file. Every stage was written from scratch, with no parser generator and no compiler framework. The converter is the last stage of the frontend: by the time it runs, the AST is complete, resolved and typed, so its job is purely to print the equivalent C code.

The generated file starts with a small boilerplate that defines the Pax primitive types as C typedefs (`u64`, `s64`, `float32` and the rest). After that come the struct definitions in dependency order, the function prototypes, the global variables, the function bodies and finally `"main"`. Since Pax does not need forward declarations, the converter emits structs topologically sorted so the C compiler sees them in the correct order. The whole file is written in one fixed sequence, shown in Figure 5.

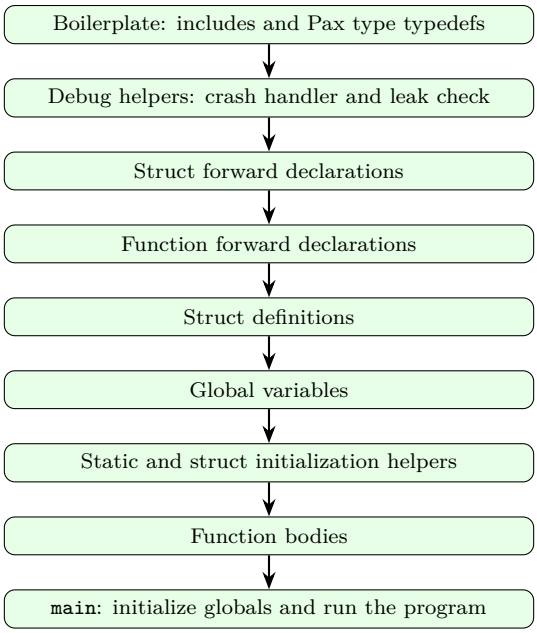



Figure 5: Order in which the C converter emits the generated C source file.

### 4.7.1 Emitting Debug directive in C

The converter also makes the generated code debuggable. In debug mode, before every statement it emits a `#line` directive that tells the C compiler where the code really came from. The Pax code in Listing 35 produces the generated C in Listing 36.

Listing 35: Pax source behind the directives

```
1 main :: () {
2     i := 0;
3     while i < 10 && (i != 5 || false) {
4         i = i + 1;
5         printf("i = %d\n", i);
6     }
7 }
```

Listing 36: Generated C with debug directives

```
1 #line 5 "C:/pax/tests/demo_if_and_while.pax"
2     s64 i = 0;
3 #line 6 "C:/pax/tests/demo_if_and_while.pax"
4     while(((i < 10) && ((i != 5) || false))){
5 #line 7 "C:/pax/tests/demo_if_and_while.pax"
6         (i = (i + 1));
7 #line 8 "C:/pax/tests/demo_if_and_while.pax"
8         printf("i = %d\n",i);
9     }
```

Because of these directives, the C compiler and the debugger report `demo_if_and_while.pax` and the right line number instead of the generated C file. This is what lets us step through Pax code directly in the debugger, and it is also what the crash handler uses to print the original source location in a stack trace. The directives are only emitted when the `-debug` flag is passed to the compiler.

### 4.7.2 Pax vs C, implicit work of C converter

Since Pax is friendly and less verbose, the converter has to make many implicit things explicit in the generated C. The examples below are taken from the test programs and show how much the frontend does behind the scenes.

The first example packs several implicit things into one line: a member access through a pointer, a string literal, and an assignment. The Pax code in Listing 37 becomes the C in Listing 38:

Listing 37: Pax

```
1  stud.name = "Timmy";   // stud: ^Student
```

Listing 38: Generated C

```
1  (((*stud).name) = string{ (u64) 5, (u8*)"Timmy" });
```

The string literal becomes a `string` struct, the pointer member access is dereferenced, and the whole assignment is wrapped in parentheses.

Structs with default member values get an implicit constructor. Listing 39 shows a struct with a default string member and a nested struct, and Listing 40 shows the generated initializer:

Listing 39: Pax

```
1  Student :: struct {
2      grade: int;
3      name: string = "Default";
4      sub: Subject;
5  }
```

Listing 40: Generated C

```
1  inline void _init_Student(Student* self){
2      self->grade = 0;
3      self->name = string{ (size_t)7, (u8*)"Default" };
4      _init_Subject(&self->sub);
5  }
```

Every declaration of a `Student` in Pax becomes a declaration plus an `_init_Student` call, so the defaults are applied and the nested struct is initialized too. The programmer never writes this.

Static arrays are lowered into a hidden backing buffer that is wired up to the wide pointer. Listing 41 shows what `arr: [5]int;`

becomes in Listing 42:

Listing 41: Pax

```
arr: [5]int;
```

Listing 42: Generated C

```
int __data__arr[5];
Static_Array arr;
arr.data = (void *)__data__arr;
arr.count = 5;
```

The same happens for arrays that are globally declared in Pax, but inside an implicitly generated `__init_global_static_arrays()` function on C side that is called at the start of `main` entry point.

Multiple return values are the last example. Since C does not have them, every function with more than one return value gets a hidden struct, and every call site unpacks it, as shown in Listing 43 and Listing 44:

Listing 43: Pax

```
q, r := divide(20, 3);
```

Listing 44: Generated C

```
struct divide_ret { int _0; int _1; };
struct divide_ret __pax_uret = divide(20, 3);
int q = __pax_uret._0;
int r = __pax_uret._1;
```

These are only a few of the implicit things the converter emits. Deferred statements are moved to the end of their scope, named and default arguments are resolved into plain positional calls, `null` becomes `nullptr`, and explicit casts become C casts. This is the cost of a friendly surface: the frontend does the work once, so the programmer does not have to do it every time.

## 4.8 Memory Management with a Pool Allocator

A pool allocator is very simple. Instead of calling **`malloc`** for every small object, it calls **`malloc`** once to get a big chunk of memory and then serves small allocations from that chunk by moving a pointer forward. When the chunk is full, it gets another one. The pool never frees individual objects. Since it keeps track of all the chunks it allocated, freeing everything at the end is just one **`free`** per chunk.

This is ideal for a transpiler. During a compilation Pax constantly allocates small amounts of memory, around 20 to 100 bytes at a time, for AST nodes, type information, unresolved references and similar compiler data. All of these objects have the same lifetime: they live until compilation finishes and then they are all discarded together. Using the global heap for each one would mean thousands of separate allocations and deallocations for no reason.

Contiguous allocation is another benefit. Objects allocated from the same block are packed next to each other in memory instead of being scattered randomly across the heap. Since parsing and semantic analysis usually visit related nodes close in time, this keeps the working set of compiler data close together.

### 4.8.1 Bump Allocation

The pool is a bump allocator. It keeps two values for the active block: **`current_pos`**, the address where the next object will go, and **`bytes_left`**, how much space remains. An allocation is just pointer arithmetic: return **`current_pos`**, advance it by the aligned size, and subtract that size from **`bytes_left`**. The requested size is rounded up to the pool's alignment of eight bytes, so every returned address is aligned for any type the compiler allocates. If there is not enough space left, the pool switches to another block

first. That is the whole allocation path.

Figure 6 shows what the active block looks like in memory. Allocated objects sit one after another from the start of the block, and **current_pos** points at the first unused byte. The next allocation is placed exactly there and **current_pos** moves forward, so objects created around the same time end up next to each other. This is also why pool allocation is faster than **malloc** for this workload: there is no free list to search and no per-object metadata to maintain.

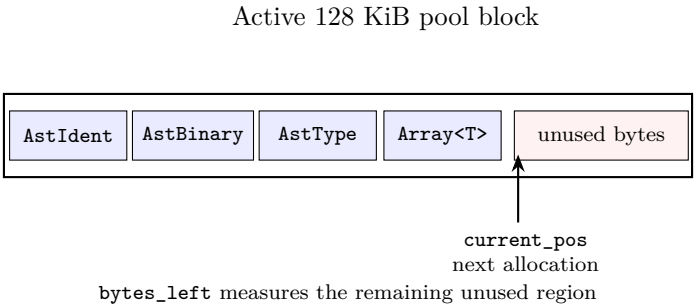



Figure 6: Bump allocation inside the active pool block. Small compiler objects are placed sequentially in the unused region, and `current_pos` advances after each allocation.

### 4.8.2 Pool Structure

Listing 45 shows everything the pool needs to remember. The member **current_memblock** is the block that is serving allocations right now, **current_pos** points at the next free byte inside it, and `bytes_left` is how much space remains. The block list holds every block the pool owns.

Listing 45: Core structure of the Pax pool allocator

```
1 struct Pool {
2     void *current_memblock = nullptr;
3     void *current_pos = nullptr;
4     size_t bytes_left = 0;
5
6     BlockList blocks;
7 };
```

The default block size is `128 * 1024` bytes (128 KiB), which is large enough to serve many small allocations before the next **`malloc`**, but small enough that a small program does not reserve several megabytes.

### 4.8.3 Block List Management

The pool does not need anything more complicated. When the current block is full, **`cycle_new_block()`** asks **`malloc`** for a new block, pushes it into the list, and makes it current. All blocks stay in the list until the end of compilation, when **`pool_release()`** walks the list and calls **`free`** on each one. There is no per-object freeing, no reuse of individual slots and nothing else to keep track of: every object shares the same lifetime, so freeing the whole blocks frees everything.

The whole allocator fits in a few lines, shown in Listing 46. Allocation is a bump with a fallback to a fresh block, and releasing the pool is just one loop over the block list.

Listing 46: Pool allocator logic (simplified)

```
pool_alloc(size):
    size = align(size, 8)           // round up to alignment (8 bytes)

    if bytes_left < size:           // if current block is full
        new_block = malloc(memblock_size) // ask malloc for a new block
        blocks.push(new_block)      // remember it for the final release
        current_block = new_block   // current working block is this one
        bytes_left = memblock_size  // reset to new size

    result = current_pos            // bump allocation
    current_pos = current_pos + size
    bytes_left = bytes_left - size
    return result

pool_release():     // at the very end
    for block in blocks:  // free all blocks at once
        free(block)
```

### 4.8.4 Custom Array Implementation

The Pax transpiler also uses a custom `Array<T>` container. It stores elements in contiguous memory and obtains storage from the global pool allocator. It is used for structures like AST statement lists, unresolved reference queues, type instantiation members, parser collections, and scope stacks.

Listing 47: Array struct for the Pax transpiler

```
template<typename T>
struct Array
{
    T *data;
    long count;
    long capacity;

    Pool *pool;

    Array(Pool *p = nullptr)
        : data(nullptr), count(0), capacity(0), pool(p) {}

    void push_back(T value);
    T get_back();
    void pop_back();

    void release();
};
```

Like standard dynamic arrays, `Array<T>` stores its elements in a contiguous sequence. It maintains a logical element count and a capacity. When the current capacity is exhausted, `push_back` allocates a larger contiguous region from the pool, copies existing elements into it, and updates the data pointer.

Listing 48: Growing a pool-backed dynamic array

```
if (count >= capacity) {
    long new_cap = capacity ? capacity * 2 : 4;

    T *new_data =
        (T *)pool_alloc(pool, sizeof(T) * new_cap);

    if (data) {
        memcpy(new_data, data, sizeof(T) * count);
    }
```

```
10
11      data = new_data;
12      capacity = new_cap;
13  }
14
15  data[count++] = value;
```

Old array buffers are not individually freed after a growth operation. This is intentional, they remain part of the pool and are released automatically when the pool is reset. The cost is a small amount of temporary memory that remains occupied till the pool reset.

Contiguous storage is important beyond allocation convenience. It allows indexed access in constant time, provides predictable sequential iteration, and improves locality when the compiler walks collections of statements, arguments, members, or unresolved references.

### 4.8.5 Memory Profiling

The pool contains optional allocation tracing for inspecting how compiler memory is used. When tracing is enabled with the `-profile` flag, every allocation records a sequence number, the block address, block size, offset within the block, requested allocation size, alignment padding, type name, returned pointer, compiler phase, and source location.

The trace is written as a pipe separated text file, one line per allocation. This alone is useful: it shows the allocation order, which AST node types consume the most space, and which Pax source location caused each allocation. However, a text file with thousands of lines is hard to read, so a small Python script converts it into a JSON file that Perfetto can display as a timeline.

Perfetto [6] is a third party performance tracing tool with a web UI. Loading the generated JSON into Perfetto renders every allo-

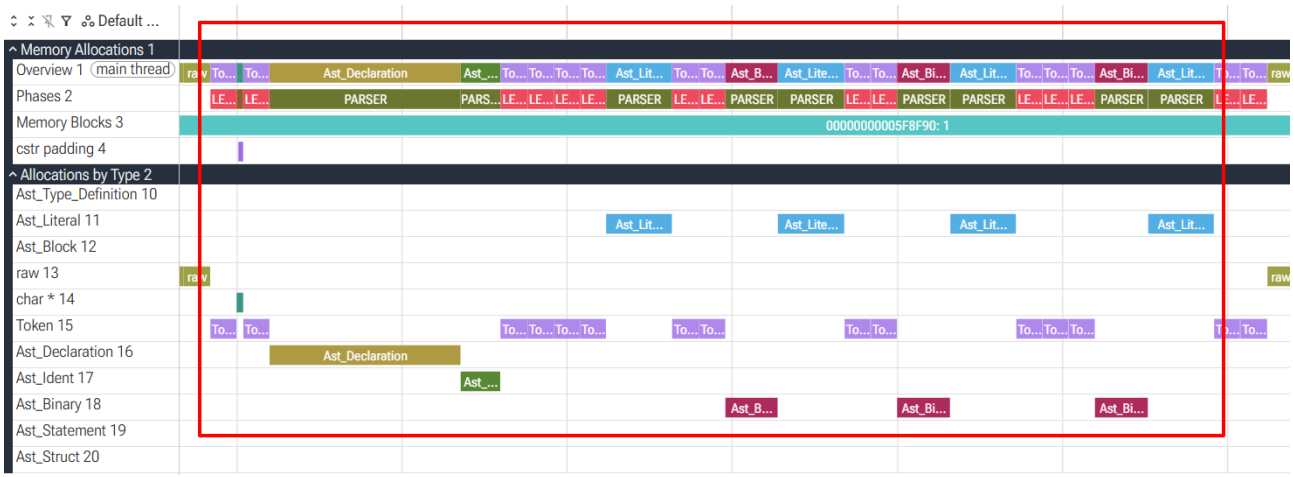



Figure 7: Perfetto timeline of an exemplary expression. Each row is a different view of the same allocations.

cation as a bar on the timeline, grouped into tracks: an overview of all allocations, one track per compiler phase, one for the memory blocks themselves, one for alignment padding, and one track per allocated type.

In Figure 7, the red outline marks the allocation for the statement:

$$a := (1 + 2 * 4)/2;$$

We see there is `Ast_Declaration` because the statement above is a declaration and its phase is `PARSER` (see row 2), then its followed by `Ast_Ident` type, it is for identifier `‘a‘`. We then see 4 `Ast_Literal` type allocations (blue) because of the 4 number literals 1, 2, 4 and 2 in the RHS of the declaration above. We also see three `Ast_Binary` types because of the operators `‘+‘`, `‘*‘`, `‘/‘`.

Moreover, we can also notice that Lexer and Parser phases one after the other since they work in parallel.

The timeline makes it easy to spot problems at a glance. The memory block track shows how the block is being filled, the phase track shows in which stage of the compiler each allocation hap-

pened, and the per-type tracks make it obvious which node types dominate. The padding track is also useful. It shows how many bytes are wasted on alignment, which helps when deciding whether a node could be laid out more tightly.

### 4.8.6 Pool vs Malloc Performance comparison

The pool fits the Pax frontend well because all compiler objects share the same lifetime: they are allocated during compilation and discarded together. A general-purpose heap allocator is designed for objects that are freed in arbitrary order, which the transpiler does not need for AST nodes and semantic data.

The advantages of the pool are:

- **Low allocation overhead:** most requests only advance a pointer inside an existing block.
- **Bulk deallocation:** all objects are released by freeing a few whole blocks.
- **Lower metadata overhead:** small nodes do not need their own global-heap allocation record.
- **Improved locality:** objects created around the same time end up next to each other.
- **Less fragmentation:** the heap sees a small number of large blocks instead of many tiny allocations.

Locality should not be overstated. An AST is a tree, so a parent node may be allocated after its children and the two are not necessarily adjacent. Still, everything lives in a few dense blocks, which is more cache friendly than objects scattered across the heap.

To measure the difference, both allocators were given 10 million AST nodes to allocate and then release. Figure 8 shows it.

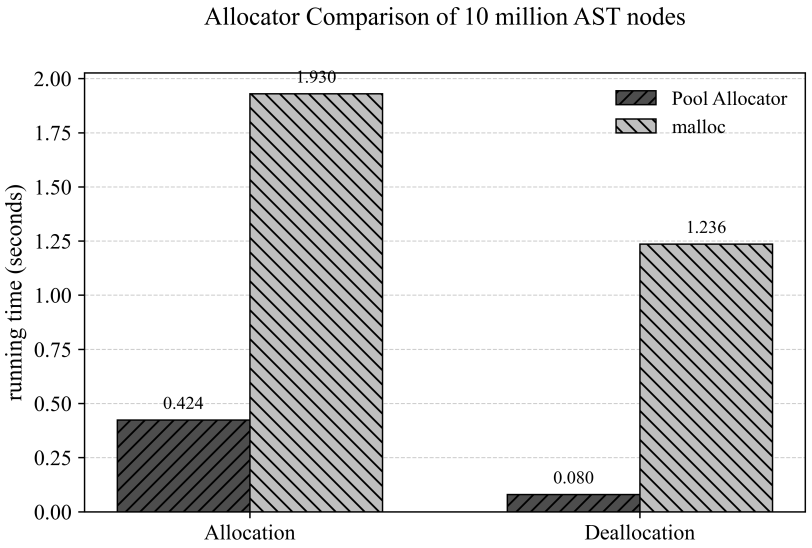



Figure 8: Allocation and deallocation time for 10 million AST nodes.

The pool allocated the nodes in 0.424 seconds while `malloc` took 1.930 seconds, so allocation was about 4.6 times faster. Releasing was even better: 0.08 seconds versus 1.236 seconds, roughly 15.5 times faster. Overall the pool finished in 0.504 seconds against 3.166 seconds, about 6.3 times faster.

The speedup is not only from skipping individual `free` calls. Allocation itself is cheaper because each request is just pointer arithmetic, and the heap only sees a few large block allocations instead of ten million small ones.

## 4.9 Error messages

Error messages are the user interface of the compiler. Without good error messages it is difficult to find bugs, especially while writing the transpiler itself. Every AST node keeps its source location, which helps a lot when debugging the compiler, because any error can be traced back to exact Pax line.

So what are good error messages? They should be short but informative and should try to pin point exactly which line and column the error is on. However, it is not always clear which part has an

error, but the compiler's job is to be as helpful as possible. Every message carries the source file path, row and column, and the compiler reprints the exact source line with the character highlighted. It is important since this way errors always point at the original Pax source and not the generated C file the user never wrote.

### 4.9.1 Error output format

Every error is thrown in the form of `"filepath[row:col]:error message"`, then the exact source line. Figure 9 shows an example. This is produced when an `u32` variable is initialized from a `s64`.

```
C:/pax/tests/d.pax[6:2]: Type mismatch in initializer for 'b', Expected 'u32' Got 's64'
    b : u32 = a;
```

Figure 9: The standard error format.

### 4.9.2 Four phases that throw errors

In the Pax transpiler there are four phases where errors can be thrown. During lexing, parsing, resolution and type inference.

**Lexer errors** are the simplest. The lexer sees an invalid character sequence, prints an error with the row and column, and exits. There is no error recovery here because invalid tokens usually make rest of file unparsable anyway. Typical messages are *"unterminated string literal"* and *"Integer literal overflow"*.

**Parser errors** occurs when the stream of tokens do not match the language grammar. For example when a semicolon is missing, when there was an 'else' without an 'if' statement, expected identifier after 'for' keyword, or there was an unexpected token at start of statement. The parser will try to synchronize to the next statement and keep going, so we can find all syntax errors.

**Resolution errors** happen when resolving identifiers, function

calls, struct types and member accesses. This is where “use of undeclared variable”, “call to undeclared function”, ”variable already declarations” can be found.

**Type inference errors** are last to happen when all the previous phases have been successful. In here, every expression receives its type and assignments, calls, returns and initializers are checked. Messages like *"Type mismatch for argument x in call to 'add'. Expected 'int' got 'string'."* name the exact parameter and both types.

### 4.9.3 Examplary errors thrown

The following screenshots are real output from the compiler. Figure 10 shows a variable used but never declared, Figure 11 a wrong argument type at a call site, and Figure 12 a missing semicolon.

```
C:/pax/tests/d.pax[6:12]: Use of undeclared variable 'a'

    b : u32 = a;
```

Figure 10: Use of undeclared variable

```
C:/pax/tests/d.pax[4:7]: Type mismatch for argument a in call to 'func'. Expected 'int', Got 'string'

   func("test");
```

Figure 11: Wrong argument type in function

```
C:/pax/tests/d.pax[6:7]: Expected ';' after variable declaration

    a := 100
```

Figure 12: A parser error

### 4.9.4 Error recovery

What is error recovery? One way of error recovery is synchronization. This is strategy used during parsing. When statement fails to parse, the parser discards tokens until it reaches a point where parsing can safely restart, like the next statement or closing braces, and then continues. It is needed because we do not want to just throw the first error and exit. We want to keep assessing further

to see if there are any more errors in the program. Without this, a single missing semicolon would cascade into dozens of misleading errors.

### 4.9.5 Deferred error reporting

Instead of printing errors immediately when they happen, Pax buffers them during each phase. The parser, resolver and the type checker each collect their errors into an Array and print them at the end of the phase depending on how the user wants them printed.

By default the compiler prints only the first error in source order and stops. With the `-verbose` flag it sorts all buffered errors by row and column and then prints them in one pass. Figure 13 shows three undeclared variables reported together.

```
PS C:\pax> pax tests\d.pax -verbose
C:/pax/tests/d.pax[4:7]: Use of undeclared variable 'missing_one'

    a := missing_one;


C:/pax/tests/d.pax[5:7]: Use of undeclared variable 'missing_two'

    b := missing_two;


C:/pax/tests/d.pax[6:7]: Use of undeclared variable 'missing_three'

    c := missing_three;


3 errors in code manager. Exiting.
```

Figure 13: Multiple errors reported with "`-verbose`" cli argument.

Each phase also finishes with a short summary line, like *"Errors in parser. Exiting."* or *"N errors in code manager. Exiting."*, so it is always clear which stage of the compiler rejected the program. This is extremely handy compared to how verbose C++ error reporting can be.

# 5 Proof of concepts

Previous chapters described the features of Pax and what it is capable of, demonstrating that it is more than an experimental language. This chapter goes a step further and shows that we can actually write and build non trivial working programs in Pax. Each program below is a complete Pax program to prove set of features, however this is only small part of the Pax test suite, there are up to 50 test cases (both that should pass and should fail) with a total of over 2500 lines of pure Pax code (excluding blank spaces and comments). This goes to show that I have genuinely stress tested the compiler and language.

The important thing to understand is that proving a compiler works is not an easy task. Since Pax is meant to be Turing complete, there is no practical limit to the edge cases a program can contain; each must be thought of manually and tested against the compiler. The test suite therefore checks both directions: that unusual combinations of features still run correctly, and that invalid programs are rejected by the frontend before any C is generated.

The purpose of the programs below is not to explain every line of code. The purpose is to show that Pax features still work when they are mixed together in longer code. It is easy to make a small example where one function is declared after it is used. It is harder when functions, globals, structs, imports and arrays are scattered through a larger program and depend on each other in different ways. These examples are written in that way on purpose.

The visual programs use the same terminal rendering pattern. A complete frame is built in a global buffer and printed with one foreign `printf` call. The frame rate is controlled by reading `__rdtsc()` and waiting until `CYCLES_PER_FRAME` has passed. This is ordinary

Pax code using foreign-function support, not a special rendering feature built into the compiler.

## 5.1 Real time ASCII raymarcher

The program below is a small terminal renderer. It draws an 80×40 sphere using ASCII characters, moves the sphere closer and away from locked camera, and does this at 20 fps. Every frame is calculated again from scratch.

The program is split into two parts. In Listing 49, `main` function is responsible for the animation itself. It keeps track of the current frame, measures time using simple `__rdtsc()` (a real time stamp counter), renderer builds the next image frame into a frame buffer, then prints the completed frame buffer and waits until the next frame should start.

Listing 49: main loop of ascii 3d sphere

```
main :: () {
    init_gradient();

    frame := 0;
    program_start := __rdtsc();

    while frame < FRAMES {
        frame_start := __rdtsc();

        info: FrameInfo;
        info.index = frame;

        elapsed_cycles := __rdtsc() - program_start;
        info.time = cast(float)elapsed_cycles / cast(float)CPU_HZ;
        info.phase = info.time * TAU;

        render_frame(info);

        clear_screen();
        printf("%s", cast(^u8)frame_buf.data);
        printf("\nframe = %d\n", frame);

        frame = frame + 1;

```

```
25          while __rdtsc() - frame_start < CYCLES_PER_FRAME {
26          }
27      }
28  }
```

The interesting work happens in `render_frame` in Listing 50. It starts by placing the camera, sphere and light for the current frame. Then it builds the camera directions needed to look at the scene. Finally, it walks over every terminal cell and checks if a ray starts at the camera and goes through this cell, does it hit the sphere and if it does, how bright should that point be.

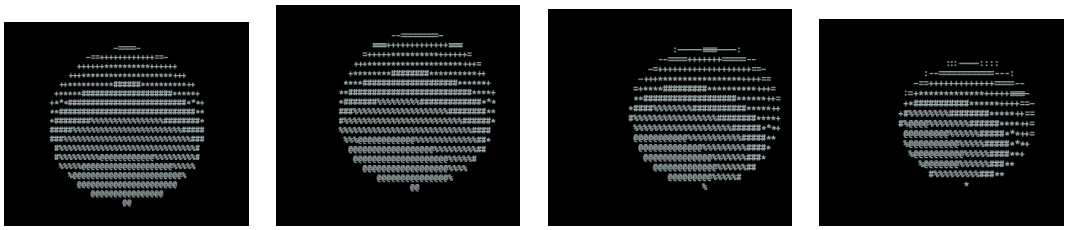
Listing 50: Building one complete frame of the raymarcher

```
1  render_frame :: (info: FrameInfo) {
2
3   t: float = cast(float)info.index / FRAMES;
4   angle := t * TAU;
5
6   // Move the camera around the sphere.
7   cam_x := 3.0 * cos(angle);
8   cam_z := 3.0 * sin(angle);
9   cam_y := 1.0;
10
11  // Move the sphere and rotate the light.
12  g_sx = 0.7 * cos(angle * 6.0);
13  g_sy = 0.0;
14  g_sz = 0.7 * sin(angle * 6.0);
15
16  g_lx = cos(angle) * 0.7;
17  g_ly = 0.8;
18  g_lz = sin(angle) * 0.7;
19
20  // Build the camera directions.
21  dir_x := -cam_x;
22  dir_y := -cam_y;
23  dir_z := -cam_z;
24  normalize3(dir_x, dir_y, dir_z, &dir_x, &dir_y, &dir_z);
25
26  up_x := 0.0;
27  up_y := 1.0;
28  up_z := 0.0;
29
30  side_x := dir_z * up_y - dir_y * up_z;
31  side_y := dir_x * up_z - dir_z * up_x;
```

```
  side_z := dir_y * up_x - dir_x * up_y;
  normalize3(side_x, side_y, side_z, &side_x, &side_y, &side_z);

  up_x = side_y * dir_z - side_z * dir_y;
  up_y = side_z * dir_x - side_x * dir_z;
  up_z = side_x * dir_y - side_y * dir_x;
  normalize3(up_x, up_y, up_z, &up_x, &up_y, &up_z);

  // Create one ray for every terminal cell.
  y := 0;
  pos: s64 = 0;

  while y < HEIGHT {
    x := 0;
    while x < WIDTH {
      u: float = cast(float)x / (WIDTH - 1) * 2.0 - 1.0;
      v: float = cast(float)y / (HEIGHT - 1) * 2.0 - 1.0;
      ray_x := dir_x + side_x * u + up_x * (-v);
      ray_y := dir_y + side_y * u + up_y * (-v);
      ray_z := dir_z + side_z * u + up_z * (-v);
      normalize3(ray_x, ray_y, ray_z, &ray_x, &ray_y, &ray_z);
      brightness := raymarch(cam_x, cam_y, cam_z,
                             ray_x, ray_y, ray_z);
      brightness = clampf(brightness, 0.0, 1.0);
      frame_buf[pos] = sample_char(brightness); // choose correct character
                                                // to emphasis brightness
      pos = pos + 1;
      x = x + 1;
    }
    frame_buf[pos] = cast(u8)10;
    pos = pos + 1;
    y = y + 1;
  }
  frame_buf[pos] = cast(u8)0;
}
```

This is why raymarcher is useful as a proof of concept. It uses structs, floats, casts, globals, nested loops, static arrays, pointers, foreign math functions, in one program.

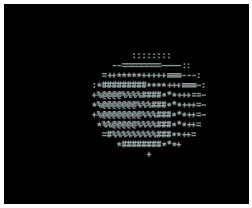
The frames in Figure 14 show that the program is actually animated. You acan try to image each frame running, here the sphere moves closer and away from camera, and the light source changes.

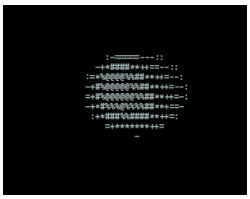

Frame 1

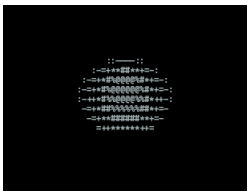

Frame 2

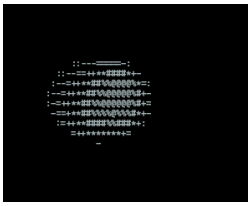

Frame 3

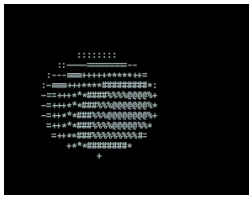

Frame 4

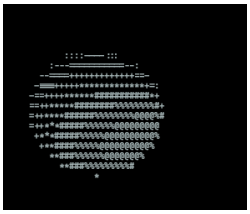

Frame 5

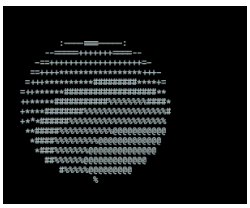

Frame 6

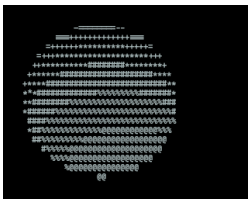

Frame 7


Frame 8


Frame 9


Frame 10


Frame 11


Frame 12

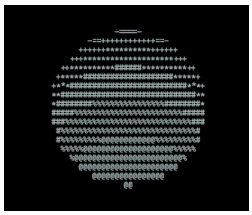

Frame 13

Figure 14: Frames of ASCII raymarcher written in Pax

## 5.2 Pluggable shaders

The program described here uses the same terminal rendering idea as raymarcher above. Only difference is it separates the renderer from the visual effect.

The visual effect is placed inside a general `shader` function. This makes it easy to write and test different terminal effects because the renderer can stay the same while the shader function is replaced.

For example, the `render_frame` function in Listing 51 calls the function `shader`.

Listing 51: The renderer only depends on the `shade` function result

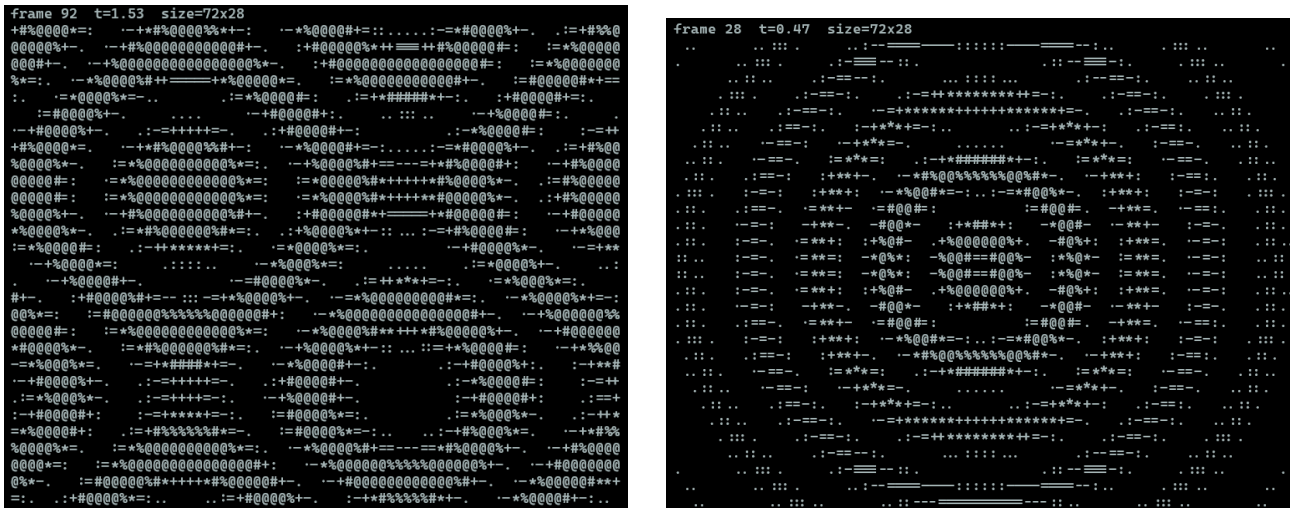
```
render_frame :: (info: FrameInfo,
                 intensity: float = 1.0,   // default parameters
                 invert: bool = false)
{
  pos := 0;
  for y: 0..HEIGHT {
    for x: 0..WIDTH {
      uv := pixel_uv(x, y);

      // implement any shader effect that follows this signature
      bright := shader(uv, info, intensity=intensity);

      if invert {
        bright = 1.0 - bright;
      }
      frame_buf[pos] = sample_palette(bright);
      pos = pos + 1;
    }
    frame_buf[pos] = cast(u8)10;  // newline
    pos = pos + 1;
  }
  frame_buf[pos] = cast(u8)0;     // null
}
```

The first version is the plasma shader. It combines horizontal and vertical waves, then turns the result into a brightness value.

Listing 52: Plasma shader

```
shader :: (uv: Vec2, info: FrameInfo,
           intensity: float = 1.0) -> float {
    t := info.phase;

    a := sin(uv.x * 3.0 + t);
    b := cos(uv.y * 3.5 - t * 1.2);
    c := sin((a + b) * 4.0);

    v := c * 0.5 + 0.5;
    return clampf(v * intensity, 0.0, 1.0);
}
```

The second version uses distance from the centre of the screen. It creates animated rings which become darker further from the centre.




**(a)** Plasma shader. **(b)** Ring shader.

Figure 15: Two different effects produced by replacing only the Pax `shader` function.

Listing 53: Animated ring shader

```
shader :: (uv: Vec2, info: FrameInfo,
           intensity: float = 1.0) -> float {
    t := info.phase;

    r2 := uv.x * uv.x + uv.y * uv.y;
    r := sqrt(r2);

    rings := sin(r * 20.0 - t * 2.5);
    fade := exp(-r2 * 0.8);

    v := (rings * 0.5 + 0.5) * fade;
    return clampf(v * intensity, 0.0, 1.0);
}
```

So nothing in `render_frame` changes between these two versions. Only the mathematical function producing brightness is replaced. This is useful because it shows that Pax can separate reusable program structure from the part that gives a program its behaviour.

Figure 15 shows the result. Both effects use the same renderer and the same terminal resolution, but they produce clearly different output.

## 5.3 Dynamic memory and pointers

To really stress test an arena allocator (which is similar but simpler to Pool allocator) was also implemented. It is the largest systems style example in demo suite. It imports the built in standard modules`String.pax` and `General.pax`. The file starts with a working arena allocator written in Pax, then continues with linked list sorting, static arrays, wide array pointers, deep pointer access, raw casts, foreign calls and numeric casts.

The arena allocator is a real allocator, not a wrapper around one call to `malloc`. It keeps the current allocation position in an `Arena` struct and keeps all backing allocations in a linked list of `Block` structs. Every request is aligned to eight bytes. If the active block has no remaining space, the allocator creates a new block and links it to the existing chain and continues

Listing 54: Using the Pax arena allocator

```
arena: ^Arena = make_arena(2048); // single arena of size 2048 bytes
defer free(arena);  // deferred free

p1 := arena_alloc(arena, 128);  // ^u8
p2 := arena_alloc(arena, 1024);
p3 := arena_alloc(arena, 512);

// request 2049 bytes which is one byte larger than the default block size
// so that it automatiically forces a larger size for new block
big: ^u8 = arena_alloc(arena, DEFAULT_BLOCK_SIZE + 1);
```

The first block has a size of 2048 bytes. The first three allocations request 128, 1024 and 512 bytes. All three values are already divisible by eight, so together they consume:

$$128 + 1024 + 512 = 1664$$

bytes of the first block.

The final allocation asks for 2049 bytes which is larger than the default block size, and it is not divisible by eight. The allocator rounds it up to 2056 bytes before creating the new block:

$$((2049 + 7)/8) \cdot 8 = 2056$$

The following code in Listing 55 is where the allocator creates the new block, links it into the arena and continues allocating from it.

Listing 55: Creating and linking a new arena block in Pax

```
if a.pos + aligned > a.current.size {
    new_size: s64 = DEFAULT_BLOCK_SIZE;
    if aligned > new_size {
        new_size = aligned;
    }

    new_blk: ^Block = malloc(sizeof(Block));
    if new_blk == null {
        return null;
    }

    new_blk.memory = cast(^u8)malloc(new_size);
    new_blk.size   = new_size;
    new_blk.next   = null;
    new_blk.used   = false;

    a.current.next = new_blk;
    a.current = new_blk;
    a.pos = 0;
}

ptr: ^u8 = a.current.memory + a.pos;
a.pos = a.pos + aligned;
return ptr;
```

## 5.4 Conway's Game of Life

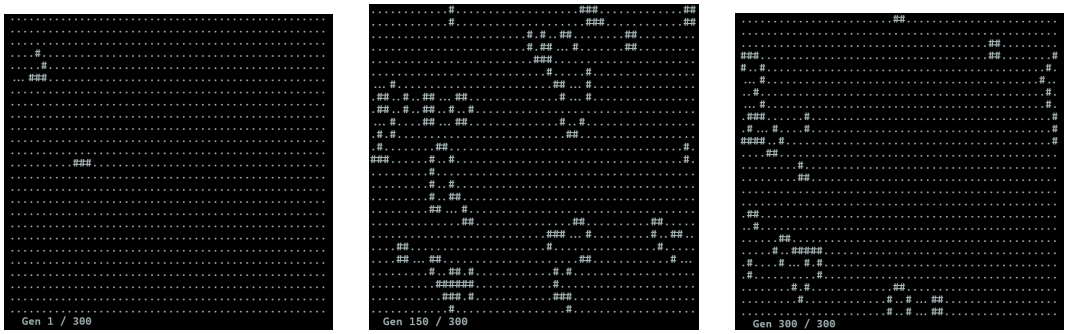
Another interesting program to implement was the classic Conway's Game of Life. The Pax version runs on a 50×25 grid for 300 generations. It begins with a glider and a blinker. The glider moves diagonally through the world, while the blinker repeatedly changes between its two states.

Unlike the terminal shaders, Game of Life keeps its state between frames. The program uses two grids, one contains the current generation and the other used to calculate the next state. After each update, the two grids swap. The grid wraps around at the edges, so cells at one edge can interact with cells on the opposite edge.

It uses two static grids passed as `^[]int` pointers, pointer swapping to move between generations, integer indexing to represent the 2D grid. The program also has to handle edge wrapping properly, so a cell on the left side sees cells on the right side as neighbours.

This is a useful proof of concept because the program has to keep updating the same state correctly for a long time. The glider and blinker have known behaviour, so they give a simple visual check that the simulation rules, grid indexing and buffer swapping work as intended.

See Figure 16 which shows 3 frames of the entire generation. First frame is the initial state where two type of patterns are placed, glider and blinker. Then a frame of middle of the generation and then 300 generation frame.




**(a)** Generation 1 **(b)** Generation 150 **(c)** Generation 300

Figure 16: Couple frame generations of Conway's Game of Life implemented in Pax. The glider moves diagonally while the blinker continues to oscillate.

## 5.5 Built-in Pax modules

As mentioned multiple times that there are built in modules written in Pax itself. The most important modules currently included with Pax are `General.pax`, `String.pax` and `Sort.pax`.

`General.pax` provides the low level functions used by other modules. `String.pax` provides heap string operations, also described earlier briefly in Section 3.2.2. `Sort.pax` provides quicksort and mergesort for integer arrays (since there is no polymorphism implemented in Pax yet).

`General.pax` declares foreign functions like `malloc`, `free`, `memcpy`, `sin`, `cos` and `pow`. Other Pax modules can import `General.pax` and use these functions without repeating their foreign declarations. See in Listing 56

Listing 56: Part of the foreign interface in `General.pax`

```
1 malloc :: (size: s64) -> ^void #foreign;
2 free :: (ptr: ^void) #foreign;
3 memcpy :: (dest: ^void, source: ^void, count: s64) #foreign;
4
5 pow :: (x: float, y: float) -> float #foreign;
6 sin :: (x: float) -> float #foreign;
7 cos :: (x: float) -> float #foreign;
8
9 sizeof :: (T: Any) -> s64 #foreign;
```

`Sort.pax` imports `General.pax` and provides two different sorting algorithms. The sorting demo imports both modules, creates one static array and copies it twice. One copy is sorted using quicksort and the other using mergesort.

Listing 57: Using the built-in sorting module

```
1 #import "General.pax"  // using General.pax
2 #import "Sort.pax"     // using Sort.pax
3
4 main :: () {
5     {  // quick sort example
6         arr: [8]int;
```

```
7           arr[0] = 8; arr[1] = 3; arr[2] = 7; arr[3] = 4;
8           arr[4] = 1; arr[5] = 9; arr[6] = 2; arr[7] = 6;
9
10          printf("Original: ");
11          print_array(&arr);
12
13          // quick sort
14          qarr := arr; // copy by value
15          quick_sort(&qarr);
16          printf("Quick sorted: ");
17          print_array(&qarr);
18      }
19
20      printf("\n");
21
22      { // merge sort example
23          arr: [8]int;
24          arr[0] = 3; arr[1] = 7; arr[2] = 2; arr[3] = 4;
25          arr[4] = 1; arr[5] = 9; arr[6] = 8; arr[7] = 6;
26
27          printf("Original: ");
28          print_array(&arr);
29
30          // merge sort
31          marr := arr;  // copy by value
32          merge_sort(&marr);
33          printf("Merge sorted: ");
34          print_array(&marr);
35      }
36
37      printf("DONE\n");
38  }
```

Both functions receive a `^[]int` array pointer. The array count is already available through `arr.count`, so the caller does not need to pass a separate length value. The output in Listing 58 shows output of the program above which produce the same sorted sequence.

Listing 58: output of demo_sortṗax of quicksort and mergesort

```
1  Original: 8 3 7 4 1 9 2 6
2  Quick sorted: 1 2 3 4 6 7 8 9
3
4  Original: 3 7 2 4 1 9 8 6
5  Merge sorted: 1 2 3 4 6 7 8 9
6  DONE
```

`String.pax` shows that higher level functionality like heap strings where strings are wide pointers can be build on top of same foreign memory functions and can be used as standaard module. In Listing 59 small fragment of code is chosen to show simple create and destroy of string that lives on heap.

Listing 59: Creating and destroying a heap string in `String.pax`

```
1  // take a string literal (which is readonly at first) and
2  // put it on heap using malloc
3  New_String :: (src: string) -> string {
4      s: string;
5      s.count = src.count;
6      s.data = cast(^u8)malloc(s.count + 1);
7
8      if s.data != null {
9          memcpy(s.data, src.data, s.count + 1);
10     }
11
12     return s;
13 }
14
15 destroy_str :: (s: ^string) {
16     if s.data != null {
17         free(s.data);
18         s.data = null;
19         s.count = 0;
20     }
21 }
```

`New_String` creates a new owned copy of string, and `destroy_str` releases its buffer. Both functions are ordinary Pax code but inside a large set of helper function that are designed to help manipulate string. The module builds higher level operations such as concatenation, slicing, comparison and cloning on top of the same string representation.

All in all, the compiler ships with useful functionality written in Pax as built-in modules, and user programs can import this functionality using the same module system used by the examples in this chapter.

### 5.6 Negative tests

The small amount of programs that showcased above are valid Pax code that can be compiled and run. But equally important are negative tests which check if invalid Pax code is rejected correctly by Pax frontend itself.

The test suit contains files that start with `demo_fail_` which are tests containing one intentional mistake. These tests check that the compiler catches the error during parsing, name resolution or type checking, instead of leaving it for the C compiler to find later.

This way it acts like practical evidence that valid programs work and invalid programs are caught at the Pax level, before reaching the C compiler.

## 6 Conclusion

This project presented Pax, a strongly and statically typed systems programming language, together with a complete transpiler that compiles it to C. The goal was never to question low-level programming itself, but to question the friction that comes with writing it: the less code we have to write, the better. Pax sits a level above C for exactly that reason.

Pax does not hide systems programming. Pointers, manual allocation, and memory layout all remain first-class parts of the language by design. The project's goal was to reduce friction rather than take control away from the programmer, because it defeats the purpose if we hide what the machine is actually doing.

The language itself is deliberately simple and straightforward in spirit. As presented in section 3, it preserves functions, structs,

loops, and conditionals, and borrows the name-first declaration syntax from Pascal, unified into a single form that makes declaration and initialization one statement, with type inference only when the type is apparent. Around that core sit the features that remove the friction. Pointer types are written as a prefix on the type they modify, unlike C, where pointers do not distribute when several variables are declared on one line, a recurring source of confusion for new users. The member access operator is a single token that hides the dereferencing required for both values and pointers, even at arbitrary levels of nesting.

On the semantic side, strings and arrays combine a length and their data as ordinary structs (Section 3.2.2) and never decay to bare pointers, removing an entire class of buffer errors along with the need for sized parameters. The defer statement schedules cleanup actions to run when a scope exits, so releasing a resource sits right beside acquiring it. Declarations are order-independent (Section 3.2.1): functions and structs can be used before they are defined, and the compiler propagates dependencies to sort struct definitions correctly. The module system takes an entire project from its entry point into one tree and emits a single C file, eliminating headers, forward declarations, and Make or CMake. Pax can call existing C libraries directly through foreign function declarations, and in debug mode the generated code ships with a runtime crash handler that traces crashes back to the original Pax source line. None of these concepts are new; the contribution is that they are designed and implemented together, and tested in a real working compiler frontend.

The frontend was implemented fully from scratch, without external libraries. A lexer creates tokens that preserve source position. A recursive descent parser handles declarations and statements, while a separate Pratt parser handles expressions, keeping the expres-

sion code short through a single precedence table. The result is an abstract syntax tree that preserves hierarchical information across phases. Because Pax allows use before definition, symbol resolution works in two passes (Section 4.6): unresolved references are deferred together with their original scopes and retried once every declaration is known. A final converter traverses the fully resolved tree and prints the single resulting C file, making the language's implicit conveniences explicit: inserting dereferences, generating default struct initializers, and unpacking return values into hidden structs on the C side.

Memory in the transpiler is managed by a custom pool allocator rather than the general-purpose heap (Section 4.8). Because compiler objects, mainly AST nodes, all live as long as the compilation, the pool allocates by bumping a pointer through large blocks and frees everything at once when compilation ends. In a test allocating 10 million nodes, allocation was roughly 4.6× faster than malloc and deallocation roughly 15.5× faster, improving overall program performance by about 6.3×. The pool also has a tracing mode that visualizes compiler heap usage over time; this was invaluable for identifying which node types consume the most space and where alignment padding is wasted.

The important distinction is that Pax is not a syntax wrapper. The frontend checks scopes, resolves identifiers, infers types, and rejects invalid code, so the C compiler is used purely for optimization and machine code generation. If Pax code compiles, the generated C is expected to compile too; any error from the C compiler is a bug in the transpiler itself. Every error points to the source the user actually wrote, and because the generated C contains line directives referencing the Pax files, debuggers and runtime crash traces do too.

The language and frontend were validated with more than 50 test

files totaling over 2,500 lines of Pax code, split between programs that must compile and negative tests that must be rejected. The proof-of-concept programs, presented in section 5, are deliberately long and combined: a real-time ASCII ray marcher animating a lit sphere at 20 fps, pluggable terminal shaders where only the mathematical effect function changes while the renderer stays identical, dynamic memory programs covering linked lists, arena allocation, and heap strings, Conway's Game of Life, and standard library modules such as String written in Pax itself. These tests show that the features do not just work in isolation, but together, in programs that do something meaningful. The suite also protected development itself: since Pax is meant to be Turing complete, a change to one feature should never silently break an unrelated one, and the test files made such errors easy to catch and fix. The complete BNF grammar of the language is provided in Appendix A.

**Advantages:** Pax's main strengths, as argued throughout this paper, are its low-level model without C's recurring friction, a fully self-contained frontend that reports every error in the user's own source, and single unit compilation that removes headers and build systems entirely.

**Disadvantages:** There are, of course, drawbacks. First, compilers are hard to write, and it is even harder to actually prove that a language works in every possible case (because Pax is meant to be a Turing complete language), since all edge cases must be thought of manually; some likely remain. Second, the crash handler currently relies on the Windows API, so it does not carry over to other platforms. Third, the compiler still uses C as its backend, which means the language must conform to what C can express, limiting the features that can be implemented.

## Future work

The next direction for Pax is a native compiler backend. The current frontend already does the heavy lifting and provides the base needed for an intermediate representation and an eventual native LLVM IR or x64 converter. Another future direction is making better use of the stored type information. The compiler already tracks the types of all symbols and expressions, which can later drive automatic struct layout and packing, as well as a Pax-native implementation of `printf` with automatic type specifiers. The standard library can also grow further. `General.pax`, `Sort.pax`, and `String.pax` are a good start, so more modules can be added and implemented in Pax itself.

# References


[1] Sean T. Barrett. *A first look at the Stu programming language*. Sept. 27, 2005. URL: https://nothings.org/computer/stu/stu.html (visited on 12/2025).

[2] Jonathan Blow. *A Programming Language for Games, talk #2*. Sept. 27, 2014. URL: https://youtu.be/5Nc68IdNKdg?t=1668 (visited on 12/2025).

[3] ISO/IEC JTC1/SC22/WG21. *Technical Report on C++ Performance (ISO/IEC TR 18015)*. See in particular sections on templates and compilation time. June 30, 2006. URL: https://www.open-std.org/jtc1/sc22/open/n3646.pdf (visited on 07/2026).

[4] Rob Pike. *Go's Declaration Syntax*. July 7, 2010. URL: https://go.dev/blog/declaration-syntax (visited on 12/2025).

[5] Vaughan R. Pratt. "Top Down Operator Precedence". In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Accessed December 2025. ACM, 1973. DOI: 10.1145/512927.512931.

[6] The Perfetto Authors. *Perfetto UI*. URL: https://ui.perfetto.dev/ (visited on 08/2026).

# A Pax Grammar in Backus–Naur Form

This appendix presents the context-free grammar of the Pax programming language using Backus–Naur Form (BNF). Non-terminal symbols are enclosed in angle brackets. Terminal symbols are written in quotation marks. The notation * denotes zero or more repetitions, and ? denotes an optional element.

```
<program> ::= <top-level>*

<top-level> ::= <import>
              | <struct-decl>
              | <func-decl>
              | <var-decl>

<import> ::= "#import" <string> ";"

<struct-decl> ::= <ident> "::" "struct"
                  "{" <struct-field>* "}"
                  ";"?

<struct-field> ::= <ident> ":" <type>
                 | <ident> ":" <type> "=" <expr>

<func-decl> ::= <ident> "::"
                "(" <param-list>? ")"
                <ret-types>?
                <func-body>

<func-body> ::= <block>
              | "#foreign" ";"

<param-list> ::= <param> ("," <param>)*

<param> ::= <ident> ":" <type>
          | <ident> ":" <type> "=" <expr>
          | <ident> ":=" <expr>

<ret-types> ::= "->" <type> ("," <type>)*

<var-decl> ::= <ident-list> ":=" <expr> ";"
             | <ident-list> ":" <type> ";"
             | <ident-list> ":" <type> "=" <expr> ";"

<ident-list> ::= <ident> ("," <ident>)*
```

```
<block> ::= "{" <stmt>* "}"

<stmt> ::= <var-decl>
         | <assign>
         | <call> ";"
         | <expr> ";"
         | <if-stmt>
         | <while-stmt>
         | <for-stmt>
         | <return-stmt>
         | <defer-stmt>
         | "break" ";"
         | <block>

<if-stmt> ::= "if" <expr> <stmt-or-block>
            | "if" <expr> <stmt-or-block>
              "else" <stmt-or-block>

<while-stmt> ::= "while" <expr> <stmt-or-block>

<for-stmt> ::= "for" <ident> ":"
               <expr> ".." <expr> <stmt-or-block>
             | "for" <ident> ":" <expr> <stmt-or-block>

<stmt-or-block> ::= <stmt>
                  | <block>

<return-stmt> ::= "return" <expr-list>? ";"

<defer-stmt> ::= "defer" <stmt>
               | "defer" <block>

<assign> ::= <lvalue> <assign-op> <expr> ";"

<assign-op> ::= "=" | "+=" | "-=" | "*=" | "/=" | "%="

<lvalue> ::= <ident>
           | <lvalue> "." <ident>
           | <lvalue> "[" <expr> "]"
           | "*" <lvalue>
           | "*" "(" <expr> ")"

<expr-list> ::= <expr> ("," <expr>)*

<expr> ::= <literal>
         | <ident>
         | <call>
         | <struct-init>
         | <unary-expr>
```

```
          | <binary-expr>
          | "(" <expr> ")"

<unary-expr> ::= ("-" | "!" | "&" | "*") <expr>

<binary-expr> ::= <expr> <binary-op> <expr>

<binary-op> ::= "+" | "-" | "*" | "/" | "%"
              | "==" | "!="
              | "<" | "<=" | ">" | ">="
              | "&&" | "||"

<call> ::= <ident> "(" <arg-list>? ")"

<arg-list> ::= <arg> ("," <arg>)*

<arg> ::= <expr>
        | <ident> "=" <expr>

<struct-init> ::= <type-name> "."
                  "{" <field-init-list>? "}"

<field-init-list> ::= <field-init>
                    | <field-init> "," <field-init-list>

<field-init> ::= <ident> "=" <expr>

<type> ::= <type-name>
         | "^" <type>
         | "[" <expr>? "]" <type>

<type-name> ::= <builtin-type>
              | <ident>

<builtin-type> ::= "void" | "bool" | "string"
                 | "int" | "float"
                 | "s8" | "s16" | "s32" | "s64"
                 | "u8" | "u16" | "u32" | "u64"
                 | "float32" | "float64"

<literal> ::= <integer>
            | <float>
            | <string>
            | "true"
            | "false"
            | "null"
```