Parallax Technical Specifications

Draft v0.1

October 30, 2024

1 Introduction

This document is the primary reference for the Parallax programming language. It officially describes the language constructs. *This document is currently in draft stage and is constantly evolving.*

2 Goals

We want to design a high-level programming language that allows easy instruction-level parallelism for its users. It is designed to be efficient and look elegant. We aim for predictive low-level performance and control over SIMD primitives in modern processors. The target architectures include AVX512 and amd64 for vector and scalar instructions, respectively.

3 Lexical Structure

3.1 Notations

The syntax of the language is given in extended BNF notation.

3.2 Encoding

All Parallax source files are in UTF-8 encoding. Other encodings are not supported. The line termination sequences can be Unix (LF), Windows (CR LF), or Macintosh (CR).

3.3 Blanks and Comments

Blanks include spaces, tabs, and form feeds. They are ignored unless separating identifiers. Comments start with '#' and run until the end of the line.

```
i = 0     # This is a comment
```

Documentation comments start with two '#' symbols and are part of the syntax tree.

4 Identifiers and Keywords

Identifiers are any string of letters, digits, and underscores but must begin with a letter and cannot end with an underscore. Case sensitivity is ignored.

$$\text{letter} ::=' A' \ldots ' Z' \mid ' a' \ldots ' z'$$

$$\text{digit} ::=' 0' \ldots ' 9'$$

$$\text{hex} ::= [-]\ (0x)\ (\text{digit} \mid ' A' \ldots ' F' \mid ' a' \ldots ' f')$$

$$\text{octal} ::= [-]\ (0o)\ ('0' \ldots ' 7')$$

$$\text{binary} ::= [-]\ (0b)\ ('0' \ldots ' 1')$$

$$\text{decimal} ::= ('1' \ldots ' 9')\ \text{digit}^*$$

$$\text{INTEGER} ::= [-]\ \text{decimal} \mid \text{hex-integer} \mid \text{octal}$$

$$\text{FLOAT} ::= [-]\ (\text{decimal}\ [.\ \text{digit}])$$

$$\text{IDENTIFIER} ::= \text{letter}\ (['\_'](\text{letter} \mid \text{digit}))^*$$

Reserved Keywords:

`and`	`break`	`const`
`continue`	`comptime`	`else`
`enum`	`except`	`export`
`for`	`fn`	`if`
`import`	`in`	`macro`
`mut`	`not`	`or`
`return`	`static`	`while`

5 Types

Parallax is a strongly-typed static language. All types are immutable by default. The `mut` keyword declares an identifier as mutable.

```
type_t [mut] identifier = expression
```

5.1 Primitive Types

We currently support the following primitive types:

- u8, u16, u32, u64
- i8, i16, i32, i64
- f32, f64
- tensor<> (more complex, harder to implement than vector/array)

5.1.1 Vectorized Types

We also support vectorized types for SIMD (Single Instruction, Multiple Data) operations. These include:

```
u.x1 , u.x2 , u.x4 , u.x8 , u.x16 , u.x32 , u.x64
i.x1 , i.x2 , i.x4 , i.x8 , i.x16 , i.x32 , i.x64
f.x1 , f.x2 , f.x4 , f.x8 , f.x16 , f.x32 , f.x64
```

Where xN indicates the number of elements in the vector. For example:

- u16x16 corresponds to a 256-bit SIMD register (YMM for x86, V register for ARM).
- u16x1 can either be padded or fit along with other values, depending on the implementation (could fit into scalar registers such as rax).

5.2 Character and String Types

5.2.1 Character Type (char)

A char is defined by the following grammar:

$$\texttt{char} \rightarrow' (' \mid \backslash i \mid \texttt{space} \mid \backslash(\texttt{[a-zA-Z]} \mid \texttt{ascii} \mid \texttt{decimal}))'$$

5.2.2 String Type (string)

A string is defined as:

$$\texttt{string} \rightarrow \texttt{"}\{\backslash i \mid \texttt{space} \mid (\texttt{[a-zA-Z]} \mid \texttt{ascii} \mid \texttt{decimal} \mid \texttt{gap})\}\texttt{"}$$

5.3 SIMD Registers for Vectorized Types

For SIMD operations, different registers are used based on the type and architecture:

- **XMM Register**: 128-bit wide, used for x86 SIMD instructions.
- **YMM Register**: 256-bit wide, also for x86 SIMD (YMM is an extension of XMM).
- **V Register**: 128-bit wide, used for ARM SIMD operations.

5.4 No Pointer/Reference Types

Parallax does not have explicit pointer or reference types. Internally, all variables, objects, and functions are passed by reference. This behavior might be extended later with something akin to Rust's `clone()` to explicitly pass by value.

5.5 User-defined Structs

Users can define their own structs to group and organize data.

6 Operators and Expressions

Unary and binary operators follow a precedence order. Statements end with semicolons.

7 Grammar Rules

7.1 Translation Unit

$$\langle\text{translation-unit}\rangle ::= \{\langle\text{external-declaration}\rangle\}^*$$

7.2 External Declaration

$$\langle\text{external-declaration}\rangle ::= \langle\text{function-definition}\rangle \mid \langle\text{declaration}\rangle$$

7.3 Function Definition

$$\langle\text{function-definition}\rangle ::= \texttt{fn}\ \langle\text{IDENTIFIER}\rangle\ (\{\langle\text{parameter}\rangle\}^*)\ \{\{\langle\text{statement}\rangle\}^*\}$$

7.4 Declaration

$$\langle\text{declaration}\rangle ::= \langle\text{declaration-specifier}\rangle\ \langle\text{declarator}\rangle\ ;$$

7.5 Declaration Specifier

$$\langle\text{declaration-specifier}\rangle ::= \langle\text{type-specifier}\rangle$$

7.6 Type Specifier

$$\langle\text{type-specifier}\rangle ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{float} \mid \texttt{double}$$

7.7 Declarator

$$\langle\text{declarator}\rangle ::= \langle\text{IDENTIFIER}\rangle \mid *\langle\text{IDENTIFIER}\rangle$$

7.8 Statement

$\langle$statement$\rangle$ ::= $\langle$expression$\rangle$; | $\langle$compound-statement$\rangle$ | $\langle$declaration$\rangle$ | $\langle$if-statement$\rangle$ | $\langle$for-loop$\rangle$ | $\langle$return-state

7.9 Compound Statement

$\langle$compound-statement$\rangle$ ::= {{$\langle$statement$\rangle$}*}

7.10 Expression

$\langle$expression$\rangle$ ::= $\langle$primary-expression$\rangle$ | $\langle$expression$\rangle$+$\langle$expression$\rangle$ | $\langle$expression$\rangle$−$\langle$expression$\rangle$ | $\langle$expression$\rangle$∗$\langle$e

7.11 Primary Expression

$\langle$primary-expression$\rangle$ ::= $\langle$IDENTIFIER$\rangle$ | $\langle$constant$\rangle$ | ($\langle$expression$\rangle$)

7.12 Assignment Expression

$\langle$assignment-expression$\rangle$ ::= $\langle$IDENTIFIER$\rangle$ = $\langle$expression$\rangle$

7.13 Parameter

$\langle$parameter$\rangle$ ::= $\langle$declaration-specifier$\rangle$ $\langle$declarator$\rangle$

7.14 If Statement

$\langle$if-statement$\rangle$ ::= `if`($\langle$expression$\rangle$)$\langle$compound-statement$\rangle$ | `if`($\langle$expression$\rangle$)$\langle$compound-statement$\rangle$ `else` $\langle$comp

7.15 For Loop

$\langle$for-loop$\rangle$ ::= `for`($\langle$declaration$\rangle$$\langle$expression$\rangle$; $\langle$expression$\rangle$)$\langle$compound-statement$\rangle$

7.16 Return Statement

$\langle$return-statement$\rangle$ ::= `return` $\langle$expression$\rangle$? ;

7.17 Constant

$\langle$constant$\rangle$ ::= $\langle$INTEGER$\rangle$ | $\langle$FLOAT$\rangle$

7.18 Integer

$\langle$INTEGER$\rangle$ ::= [−]?$\langle$decimal$\rangle$ | [−]?$\langle$hex$\rangle$ | [−]?$\langle$octal$\rangle$ | [−]?$\langle$binary$\rangle$

7.19 Float

$\langle$FLOAT$\rangle$ ::= [−]?$\langle$decimal$\rangle$["."$\langle$digit$\rangle$*]

7.20 Decimal

$$\langle\text{decimal}\rangle ::= ('1' \ldots '9')\langle\text{digit}\rangle^*$$

7.21 Hexadecimal

$$\langle\text{hex}\rangle ::= \texttt{0x}\langle\text{digit}\rangle \mid' A' \ldots' F' \mid' a' \ldots' f'$$

7.22 Octal

$$\langle\text{octal}\rangle ::= \texttt{0o}'0' \ldots' 7'$$

7.23 Binary

$$\langle\text{binary}\rangle ::= \texttt{0b}'0' \ldots' 1'$$

7.24 Identifier

$$\langle\text{IDENTIFIER}\rangle ::= \langle\text{letter}\rangle\,(\texttt{['\_']}\,(\langle\text{letter}\rangle \mid \langle\text{digit}\rangle))^*$$

7.25 Letter

$$\langle\text{letter}\rangle ::=' A' \ldots' Z' \mid' a' \ldots' z'$$

7.26 Digit

$$\langle\text{digit}\rangle ::=' 0' \ldots' 9'$$

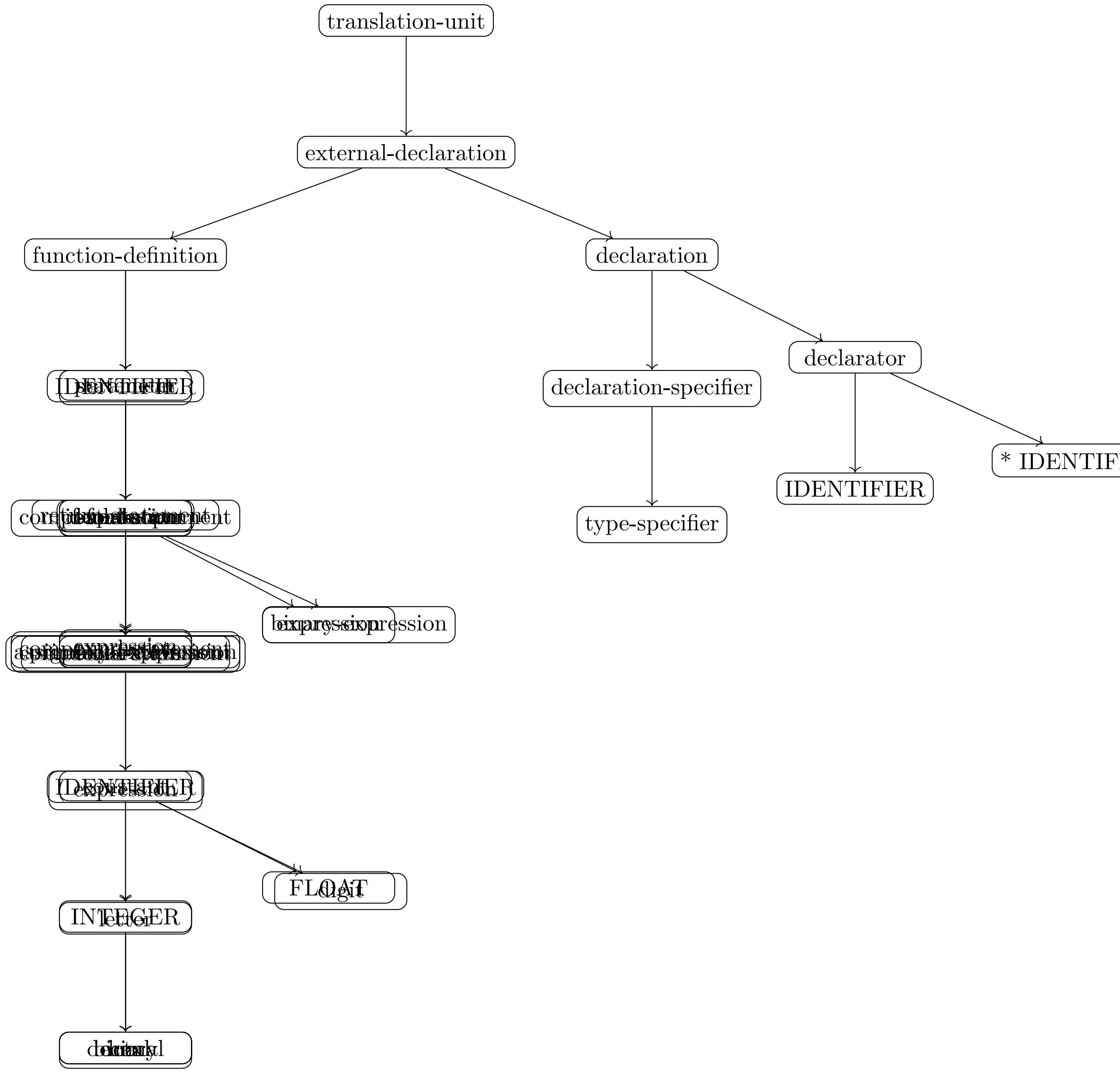
7.27 String Literal

$$\langle\text{string-literal}\rangle ::= "\langle\text{character}\rangle^*"$$

7.28 String

$$\langle\text{STRING}\rangle ::= \langle\text{string-literal}\rangle$$

7.29 Abstract Syntax Tree



8 Error Handling

We are considering a `try/catch` mechanism, possibly similar to `Result` types in Rust. Null values may be avoided for performance reasons.

9 Memory Management

We are exploring garbage collection strategies and will investigate ownership and borrowing concepts.

10 Optimizations

10.1 Constant Folding

Constant folding evaluates constant expressions at compile time, reducing runtime computations by pre-calculating known values.

10.2 Loop Unrolling

Loop unrolling replicates the loop body multiple times to reduce loop overhead. This technique can improve instruction-level parallelism and cache utilization.

10.3 Constant Propagation

Constant propagation replaces variables with their constant values throughout the code. This enables further optimizations and simplifies computations.

10.4 Dead Code Elimination

Dead code elimination removes code that does not affect the program's output, improving performance and reducing code size.

10.5 Function Inlining

Function inlining replaces function calls with the function's body, reducing call overhead and enabling further optimizations.

10.6 Instruction Scheduling

Instruction scheduling reorders instructions to maximize pipeline efficiency, improving instruction-level parallelism and CPU utilization.

10.7 Data Layout Optimization

Data layout optimization reorganizes data structures to improve cache locality, enhancing memory access patterns and overall performance.

10.8 Auto Vectorization

Auto vectorization automatically converts scalar operations to vector operations, leveraging SIMD instructions for improved parallel processing. This is the major optimization we want to research.