# Une

## Identifier Resolution in Programming Languages

Elijah Shadbolt

Department of Game Development, Media Design School, Auckland, New Zealand

### Abstract

As a game programmer, Elijah Shadbolt is interested in exploring the field of programming languages. Working as a software programmer involves reading, maintaining and debugging code written by other people. One aspect of programming languages that increases the chance of missing errors while reading code [?] is how programmers use variables bound to readable names declared in different parts of the source code, and how those named references are resolved.

We propose to design a programming language specification that eliminates the appearance of identical and ambiguous identifiers in the source code, to improve the clarity and readability for developers who did not author the code, and to mitigate uncaught errors arising from maintenance.

To achieve this, we will design the specifications for a tiny, simple, imperative programming language called Une, including its syntax and core language features, along with source code examples and a prototype compiler implementation. Elijah has had prior experience implementing an interpreter for pure lambda calculus [1].

**Keywords:** identifier resolution, name binding, programming language specifications, scope, alias

### Background

The language of C allows identifiers comprised of one or more ASCII alphanumeric characters, with the first character being a letter [?]. Element declarations (for variables, functions, and types) can be referenced from any nested scope just by name. Identifier resolution begins searching in the reference's scope, then moves upwards to each parent scope, until a declaration is found with that equivalent identifier.

```
// Fig.1
int alpha = 1;
int main() {
    int beta = 2;
    return alpha + beta; // returns 3
}
```

Fig.1 demonstrates that `alpha` is resolved as the global variable, and `beta` is resolved as the local variable.

```
// Fig.2
int alpha = 1;
int main() {
    int alpha = 2;
    return alpha + alpha; // returns 4
}
```

Fig.2 demonstrates that `alpha` is resolved as the local variable. The global variable's name has been hidden and cannot be referenced from inside `main`.

Function parameters are considered local variables in the function's body scope.

The main drawback of this design is that in a large function body, if an editor comes along and adds a new local variable with a name that hides a global variable or function, any pre-existing references to that global element in nested scopes will now resolve to the wrong element, causing compiler errors which can only be resolved by renaming the new local variable.

## Python

In python, variables are declared and defined at first assignment. This means that globals outside the function scope cannot be mutated unless it is explicitly stated that the identifier is for a global variable. Reading a global variable is as simple as using its name, just like C. Fig.3 demonstrates mutating a global variable in Python.

```
# Fig.3
x = "global"
def foo():
    global x
    x = "modified"
    print("x inside:", x)
foo()
print("x outside:", x)
# OUTPUT:
# x inside: "modified"
# x outside: "modified"
```

This design is useful in the sense that accidental mutation of global variables is prevented. However, it can cause a bug if a programmer creates a local variable when they want to mutate a global variable. The code will be interpreted without errors, thus hiding information from the person debugging the bug.

## Overloading in C++

Languages like C++, C#, and Java allow users to overload functions with different function signatures that use the same identifier. The identifier represents an overload set, and the types of the arguments provided to a function call determine which overload is used. This is a form of static polymorphism; resolved at compile time.

The best use case for overloading is when two functions have the same functionality, with slightly different types which share common traits. Another use case is for providing optional arguments. Fig.4 demonstrates overloading in C++.

```
// Fig.4
short add(short a, short b) {
    return a + b;
}
long add(long a, long b) {
    return a + b;
}
int main() {
    long c = add(3L, 4L);
    return c; // returns 7L
}
```

The identifier `add` is resolved as the function with signature `long add(long, long)`.

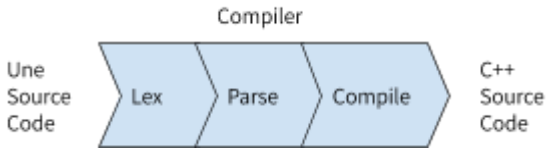
The issue of this duplication of code has been addressed by C++ in the form of function templates, which provide an overload set for any parameter types. The upcoming standard C++20 will refine templates further with TS Concepts.

If a programmer needs to make sure they are using a specific overload, they have to cast the identifier as a function pointer with the specific signature.

## Methodology

This project will primarily focus on the language specifications. It will describe the abstract syntax tree, how to parse the language, and the behaviour of programs written in the language.

The compiler implementation will be frugal; a prototype to prove that the language can be used effectively. It will be a source-to-source compiler (transpiler), which will output C++ source code files and header files, which can in turn be compiled by Microsoft's MSVC compiler.




This tiny language will have very few core language features, mostly derived from C [?]. Arithmetic and comparison operators will be carried over. Primitive types will be similar. This static typed language will make heavy use of the stack and heap pointers, and therefore manual memory management will be required [?].

| Week | Task |
|---|---|
| 1 | Language Specifications |
| 2 | Language Specifications |
| 3 | Source Code Examples |
| 4 | Lexer and Parser |
| 5 | Compiler (Transpiler) |
| 6 | Compiler (Transpiler) |

### Required Resources (Software)

- Abstract Syntax Tree Parser Tool

## Design

### Element Scopes

- ❏ Namespace
  - ❏ Namespace
  - ❏ Constant (Static Readonly Variable)
  - ❏ Function
    - ❏ Variable, Parameter
    - ❏ Alias
    - ❏ Flow Control Scope...
  - ❏ Type (Struct)
    - ❏ Instance Variable (Field)
    - ❏ Alias
  - ❏ Alias (Type, Function, Variable)

Members that can be referenced in an expression will be under the categories Type, Variable, Function, or an Alias of such a category. Functions can have nested scopes for flow control like IF statements and FOR loops. Any scope can have an Alias.

A member can be referenced simply by its name if the reference is in the same scope as the member declaration.

```
var a = "hello world"
global::print(a)
```

Referencing struct members will be done by direct access (`"obj.name"`) and pointer indirection (`"obj->name"`) [?]. Referencing static members like types, functions and constants will be done similar to C++ (`"Math::pi"`) [?].

Members in the parent scope can be referenced by adding a prefix (like `"../"` for relative file paths [?]). For example, `"$"`.

Another category of alias can be declared which enables the member to be referenced in nested scopes directly by name. This has the consequence of making it a keyword that cannot be used for new identifiers in nested scopes.

```
let pi: global::Double = 3.14
fun print_pi() -> Void
{
  keyword alias global::print
  print($pi)
  if (true)
  {
    print($$pi)
    print(global::pi)
  }
}
```

## Discussion

- Should the parent scope prefix be an operator (`"$name"`) or a keyword like the C# class instance keyword (`"this.name"`)?
- Would it be convenient to provide several built-in parent scope keywords, like “global”, “this”, and “func”? Our current opinion is yes.
- What would happen if users could label and reference regular scope blocks?
- Identifier equality irrespective of naming conventions, like Nim? [3]
- Poisoning keywords to allow re-defining names in nested scopes?
- How would this behave with preprocessor definitions and macros?
- Would it be practical to incorporate a similar feature into existing languages?

## References

[1] Shadbolt, E. (2016). *Defunct*: A Pure Lambda Calculus Interpreter. https://cresspresso.github.io/defunct/

[2] Singh, R., Sharma, V., & Varshney, M. (2009). Design and Implementation of Compiler. New Age International. http://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=285000&site=ehost-live&ebv=EB&ppid=pp_Cover

[3] Nim https://nim-lang.org/docs/manual.html#lexical-analysis-identifier-equality