Amber - nowy język programowania zbudowany na frameworku Heraclitus ze wsparciem dla podświetlania składni w VSCode oraz dokumentacją

(Amber - a new programming language built on the Heraclitus framework with syntax highlighting support in VSCode and documentation)

Paweł Karaś

Praca licencjacka

Promotor: dr Łukasz Piwowar

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

14 czerwca 2024

Streszczenie

Amber to innowacyjny język programowania, zaprojektowany z myślą o generowaniu skryptów w powłoce Bash w sposób bardziej czytelny i bezpieczny. Oparty został na nowoczesnych wzorcach składniowych inspirowanych językami JavaScript, Rust i Swift. Amber zawiera system ścisłego typowania oraz mechanizmy zarządzania błędami, co czyni go przyjaznym i wydajnym narzędziem do pisania skryptów.

W tej pracy dowiemy się jak wygląda składnia języka, w jaki sposób wcześnie sygnalizowane błędy (podczas kompilacji) mogą pomóc w tworzeniu kodu, jak wygląda gramatyka w zapisie EBNF opisująca język Amber, jakie mechanizmy zachodzą wewnątrz kompilatora, opis stworzonego przeze mnie frameworka Heraklit oraz opis wtyczki podświetlającej składnię języka dla środowiska programistycznego Visual Studio Code. Następnie dowiemy się jakie są plany rozwoju tego języka oraz przeanalizujemy jego debiut na rynku.

Amber is an innovative programming language designed with the goal of generating Bash shell scripts in a more readable and secure manner. It is based on modern syntax patterns inspired by JavaScript, Rust, and Swift. Amber features a strict typing system and error handling mechanisms, making it a user-friendly and efficient tool for writing scripts.

In this work, we will explore the language's syntax, how early error detection (during compilation) can aid in code creation, the EBNF grammar describing the Amber language, the mechanisms within the compiler, a description of the Heraclitus framework I developed, and a description of the syntax highlighting plugin for the Visual Studio Code development environment. Next, we will discuss the development plans for this language and analyze its debut on the market.

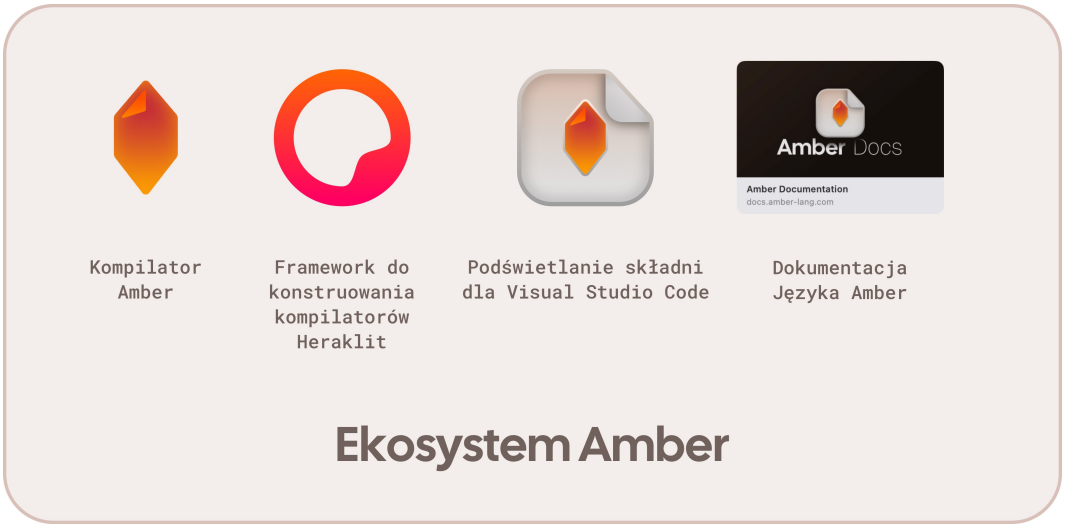
Spis treści

Rozdział 1.

Wprowadzenie

Na wstępie przedstawię ekosystem języka Amber [9]. W tym kontekście nazywam nim zbiór technologii, które razem budują całość doświadczenia programowania w języku. Składają się na niego:

- **Kompilator** [10] języka Amber
- **Framework** [11] Heraklit służący do konstruowania parserów, w którym wyżej wspomniany kompilator został napisany
- **Wtyczka** [8] dla oprogramowania Visual Studio Code do podświetlania składni tego języka
- **Dokumentacja** [7] języka Amber, która za zadanie ma wytłumaczyć użytkownikowi jak z niego korzystać



1.1. Problem

Pewnego dnia stanąłem przed wyzwaniem stworzenia pliku instalacyjnego, który byłby uniwersalny i kompatybilny z różnymi systemami operacyjnymi. Głównym celem było stworzenie instalatora, który działałby zarówno na systemie macOS, jak i na różnych dystrybucjach Linuxa. Zadanie to okazało się wyjątkowo skomplikowane z kilku powodów:

1. **Minimalna obsługa**. Celem było, aby utrzymywanie rozwiązania nie wymagało znacznych nakładów czasowych.

2. **Wysoka przenośność**. Dążyłem do uzyskania maksymalnej przenośności rozwiązania. Optymalnie jeden kod napisany w jednym języku, który mógłby być dostosowywany do różnych platform.

3. **Prosty proces instalacyjny**. Celem było, aby użytkownik nie potrzebował dodatkowych zależności (umożliwiających zainstalowanie pakietu).

1.2. Instalacja

Pierwotnie rozważałem stworzenie osobnych instalatorów dla macOS i Linuxa. Na system macOS użyłbym oprogramowania `productbuild`, które pozwala na tworzenie zaawansowanych pakietów instalacyjnych `.pkg`. W przypadku Linuxa sprawa jest już o wiele bardziej skomplikowana. Musiałbym zbudować osobny instalator dla najpopularniejszych dystrybucji. Dla tych opartych na Debianie, użyłbym `dpkg-deb` do tworzenia pakietów `.deb`. Dystrybucje oparte na Red Hat, wymagałyby podejścia z użyciem `rpm`, do tworzenia pakietów RPM. Z kolei dla Arch Linux, zakładając że zainstalowany jest menedżer pakietów Pacman, niezbędne byłoby generowanie paczek z rozszerzeniem `.pkg.tar` za pomocą `makepkg`. Ponad to musiałbym zapewnić użytkownikom możliwość ręcznej instalacji, na wypadek gdyby ich dystrybucja nie obsługiwała żadnego z wymienionych formatów.

Proces generowania instalatora dla każdej platformy wymaga utrzymywania wielu różnych manifestów instalacyjnych. Każda zmiana metadanych instalatora wiąże się z koniecznością aktualizacji wszystkich manifestów, z których każdy posiada inny format, co znacząco utrudnia wprowadzanie prostych aktualizacji przy użyciu osobnego skryptu automatyzującego. Gdyby nawet zainwestować czas w stworzenie takiego osobnego skryptu, konieczne byłoby również jego testowanie oraz utrzymywanie dla każdego projektu.

To pokazuje, że tworzenie uniwersalnego instalatora napotyka liczne wyzwania na platformie Linux. Niestety takie rozwiązanie nie spełnia wymogów `1` oraz `2`.

Platforma	Manifest	Format manifestu	Extension
macOS	`app.entitlements`	XML	`.pkg`
Debian	`control`	Debian Control file	`.deb`
Red Hat	`app.spec`	RPM spec file	`.rpm`
Arch	`makepkg.conf`	Makepkg Config file	`.pkg.tar`

Tabela 1.1: Rodzaje instalatorów dla różnych platform.

1.3. Menedżery pakietów

Inna strategia, która przyszła mi do głowy, to zbudowanie pakietu instalacyjnego dla jednego czy dwóch wspólnych menedżerów pakietów. Dla systemu macOS oraz popularnych dystrybucji Linuxa stworzyłbym *tap*[1] w języku Ruby dla menedżera pakietów Homebrew [17]. Choć niemal prawie każdy programista na systemie macOS ma zainstalowany `brew` na swoim komputerze, to niestety Homebrew nie jest tak szeroko znany i używany na różnych dystrybucjach Linuxa. Wydanie oprogramowania wyłącznie za pomocą `brew` spełniałoby wymogi `1` oraz `2`, czyli to, czego nie są wstanie zapewnić tradycyjne instalatory. Jednakże w tym przypadku nie jesteśmy w stanie spełnić wymogu `3`. Użytkownicy systemu Linux musieliby osobno zainstalować ten menedżer pakietów.

Aby zaradzić temu problemowi, należałoby wydać to oprogramowanie dla wielu różnych popularnych menedżerów pakietów Linuxa takich jak `apt`, `rpm` oraz `pacman`. W przypadku `apt` musiałbym stworzyć PPA [3] w języku Python oraz wygenerować instalator `.deb`. Dla `rpm` musiałbym wygenerować instalator `.rpm` oraz stworzyć osobne repozytorium dla tej paczki napisane w mieszance tekstowego opisu specyfikacji pakietu oraz Bash[2]. W przypadku `pacman` sytuacja się mocno komplikuje, ponieważ nie ma możliwości dodania repozytorium z moim własnym oprogramowaniem dla użytkowników. Pacman jest przeznaczony do pracy z oficjalnymi repozytoriami Arch Linuxa i nie obsługuje bezpośrednio repozytorium AUR. Częściowym rozwiązaniem problemu mogłoby być zasugerowanie użytkownikom korzystania z oprogramowania *third-party* jak `yay`, `trizen` lub `pamac`. Aby stwierdzić, popularność możliwych rozwiązań sprawdziłem ilość gwiazdek na platformie GitHub [15].

Nazwa menedżera pakietów	Liczba gwiazdek
`yay`	10 460
`trizen`	784
`pamac`	222

Tabela 1.2: Stan z 27 maja 2024

[1] Tap w terminologii Homebrew oznacza repozytorium Git zawierające dodatkowe formuły, czyli oprogramowanie możliwe do zainstalowania.

[2] Bash (*Bourne Again SHell*) - jeden z najbardziej rozpowszechnionych interpreterów poleceń na systemach Unix

Analiza wykazała, że `yay` jest najpopularniejszym rozwiązaniem na platformie Arch Linux.

Łatwo jednak zauważyć, że sytuacja w której spełnialiśmy punkty `1` i `2`, została zastąpiona sytuacją w której spełniamy jedynie wymóg `3`.

1.4. Skrypty instalacyjne

Ostatnim pomysłem, który przyszedł mi do głowy było zbudowanie skryptu instalacyjnego, który można uruchomić na każdym komputerze. Pocieszającą wiadomością jest fakt, że zarówno Linux jak i macOS mają wspólnie zainstalowany interpreter poleceń Bash.

1.4.1. Czym jest Bash

Bash [2] to najpopularniejsza powłoka w systemach Unix. Jest często stosowana jako domyślna w wielu dystrybucjach Linuxa oraz w systemach macOS.

Oprogramowanie to zostało stworzone już w 1989 roku, a jego rozwój nie wprowadza żadnych drastycznych zmian. Dzięki temu, że Bash wspiera wsteczną kompatybilność, kod napisany dla danej wersji uruchomi się również w nowszych wersjach. Zdecydowałem się wspierać wersję 2.0, która została wydana w 1996 roku. Była to jedna z bardziej przełomowych wersji Bash, która dodała kluczowe funkcje takie jak:

- **Tablice** - umożliwiające manipulację zbiorami danych w pamięci.
- **Arytmetyczna pętla for** - umożliwia wykonywanie działań arytmetycznych - `for (( a; b; c ))`.
- **Dynamicznie ładowane funkcje** - pozwalające na modularne podejście do pisania skryptów i ładowania funkcji z zewnętrznych skryptów na żądanie.

Ponadto ta wersja jest starsza niż Windows XP stworzony w 2001 roku, który przestał być wspierany w 2014 roku.

Analizując tę sytuację pod kątem systemów macOS, widać, że system operacyjny na komputerach Macintosh firmy Apple, w jego najstarszej wersji OS X 10.0 Cheetah, wydanym w 2001 roku, czyli 5 lat po premierze Bash 2.0, najprawdopodobniej korzystał z tej lub nowszej wersji powłoki systemu. Oznacza to, że nasz instalator jest kompatybilny ze wszystkimi komputerami firmy Apple.

Jeśli chodzi o systemy Linux, sytuacja jest bardziej złożona. Niemniej jednak większość nowoczesnych dystrybucji wspiera skrypty napisane w Bash 2.0. Najnowsza

wersja tej powłoki to 5.0, która została wydana w 2020 roku. Dystrybucje Linux można podzielić na następujące kategorię pod kątem częstotliwości aktualizacji:

- **Rolling Release** (np. Arch) - te dystrybucje ciągle aktualizują oprogramowanie do najnowszych wersji.
- **Regularne wydania** (np. Ubuntu) - te dystrybucje wydają nowe wersje co roku lub co pół roku.
- **Dystrybucje o długim wsparciu „LTS”** (np. Debian LTS) - te dystrybucje wydają nowe wersje raz na około 2-3 lata.

Jak widać metoda ta spełnia na pewno wymagania `2` i `3`. Jeśli chodzi o wymóg `1`, po pewnym czasie zauważyłem, że nie jest tak prosto. Okazuje się, że Bash został zaprojektowany, aby był prosty w użyciu w terminalu lecz niekoniecznie zwraca uwagę na wygodę programowania. Podczas programowania w tym języku napotkałem kilka aspektów, które sprawiły, że spędzałem mnóstwo czasu na rozwijaniu i debugowaniu skryptów:

- **Trudna składnia języka** - ze względu na to, że Bash powstał w czasach gdy standardy projektowania języków programowania jeszcze ewoluowały, jego składnia może być czasami trudna do zrozumienia i nieintuicyjna, szczególnie dla początkujących programistów.
- **Brak zabezpieczeń** - mechanizmów zapobiegających ignorowaniu zarówno pozytywnych, jak i negatywnych scenariuszy w logice przepływu sterowania.
- **Brak kontroli typów** - oznacza, że błędy związane z niezgodnością typów mogą zostać niezauważone aż do momentu wykonania konkretnego fragmentu kodu.

Okazało się, że nie istnieje język kompilowany do Bash, który spełnia wszystkie te wymagania. Zdecydowałem, że taki zaprogramuję.

Rozdział 2.

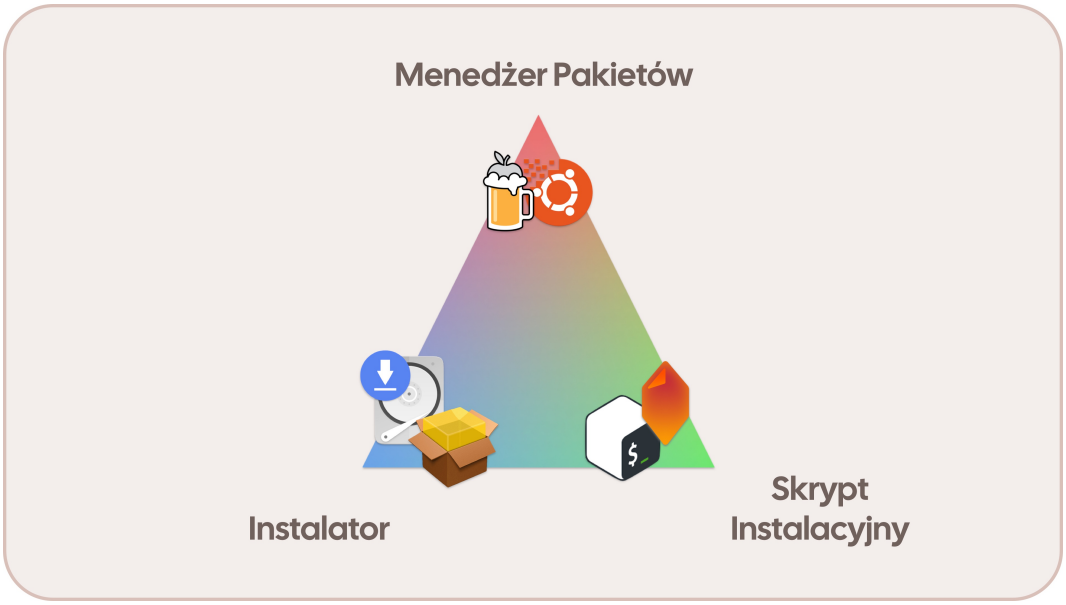
Amber

2.1. Czym jest Amber?

Amber to język programowania zaprojektowany do generowania czytelnych i bezpiecznych skryptów Bash. Jego głównymi cechami są:

- **Nowoczesna składnia** - Amber jest inspirowany językiem JavaScript (ECMAScript 2015), który jest szeroko znanym i rozpoznawanym językiem programowania. Czerpie również inspiracje z innych nowoczesnych języków takich jak Rust oraz Swift.
- **Zabezpieczenia obsługi błędów** - Amber wymusza obsługę błędów, które mogą pojawić się przykładowo przy wywoływaniu poleceń Bash.
- **System typów** - na tyle ścisły, by nie pozwalać na niedozwolone operacje w trakcie kompilacji takie jak przykładowo dodawanie tablicy z liczbą. Jednocześnie na tyle elastyczny, że programista może jednoznacznie stwierdzić typ danej wartości.

Amber znajduje się w sekcji skryptów instalacyjnych obok języka Bash w diagramie kategorii instalatorów na rysunku 2.1. Kod w języku Amber jest transpilowany do równoważnego w języku Bash. Innymi słowy można by powiedzieć, że Amber to bardziej przyjazny i bezpieczny interfejs do pisania skryptów Bash.



Rysunek 2.1: Diagram trójkątny kategorii instalatorów.

2.2. Dla kogo jest Amber?

Amber jest przeznaczony dla użytkowników, którzy chcą tworzyć skrypty Bash w sposób bezpieczny i czytelny. Obejmuje to takie grupy jak:

1. **Programiści Bash** - dla tych, którzy już tworzą skrypty Bash, ale chcą poprawić ich czytelność i bezpieczeństwo, Amber oferuje nowoczesną składnię i mechanizmy, które zapobiegają typowym błędom, co może ułatwić zarządzanie bardziej złożonymi skryptami.

2. **Programiści innych języków** - osoby zaznajomione z językami takimi jak JavaScript, Rust czy Swift, mogą znaleźć w języku Amber znane sobie konstrukcje i składnię, co ułatwi im wejście w świat skryptów Bash bez konieczności nauki składni od podstaw.

3. **Administratorzy systemów i DevOps** - profesjonaliści zajmujący się automatyzacją zadań na serwerach i w chmurze, mogą wykorzystać język Amber do efektywnego i mniej podatnego na błędy pisania skryptów niż czysty Bash.

4. **Hobbyści i uczący się** - osoby dopiero rozpoczynające swoją przygodę z programowaniem lub automatyzacją zadań w systemach Unixowych, mogą łatwiej rozpocząć dzięki bardziej intuicyjnej składni i zabezpieczeniom języka Amber.

5. **Zespoły deweloperskie** - w środowiskach, gdzie różni programiści pracują razem nad skryptami instalacyjnymi lub automatyzacją, może pomóc w utrzymaniu spójności kodu, jego bezpieczeństwa i czytelności, co jest kluczowe dla efektywnej współpracy.

Rozdział 3.

Składnia języka Amber

Przykładowy fragment kodu, który ilustruje pewne cechy języka Amber. W nadchodzących rozdziałach omówię szczegółowo każdą z nich osobno. Zauważmy między innymi instrukcję `echo`, która wypisuje wyrażenie do standardowego wyjścia. Będziemy z niej korzystać, aby testować wartości zwracane przez wyrażenia.

```
// Definiowanie zmiennych
let name = "John"
let age = 30

// Wyświetlanie powitania
echo "Cześć, mam na imię {name}"

// Wykonywanie wyrażeń warunkowych
if age < 18 {
    echo "Jeszcze nie jestem dorosły"
} else {
    echo "Jestem dorosły"
}

// Przechodzenie przez tablicę
let fruits = ["arbuz", "banan", "cytryna", "daktyl"]
echo "Moje ulubione owoce to:"
loop fruit in fruits {
    echo fruit
}
```

3.1. Typy danych

Bash w wersji 2.0 obsługuje tylko jeden natywny typ danych, którym jest ciąg znaków (jego wewnętrzna reprezentacja składa się z tablicy znaków `char*`).

Amber obsługuje pięć typów:

- `Text` - typ danych tekstowych. W innych językach programowania najczęściej określany jako *String*.
- `Num` - typ danych numerycznych. To po prostu dowolna liczba zmiennopozycyjna.
- `Bool` - typ danych logicznych *Boolean* (flaga). Może przyjmować jedną z dwóch wartości `true` (prawda) lub `false` (fałsz).
- `[]` - typ danych tablicowy.

To oznacza, że prymitywne typy danych (czyli wszystkie oprócz tablic) są przechowywane w pamięci jako tablice znaków. Sama idea istnienia różnych podstawowych typów w języku Amber została zaimplementowana głównie po to, by pomóc programiście w wykryciu podstawowych błędów. Poniekąd Amber jest podobnym rozszerzeniem jak TypeScript dla języka JavaScript.

3.1.1. Tekst

Tekst w języku Amber to ciąg znaków zdefiniowany w cudzysłowie, taki jak na przykład `"Hello world"`. Jest reprezentowany przez typ `Text`.

```
// Literał ciągu znaków `Text`:
"Ala ma kota"
```

Istnieje znak ucieczki (*escape character*) `\` umożliwiający uwzględnienie w tekście dowolnego znaku, który oznaczałby zmianę składniową. Przykładowo w `"a"b"` koniec tekstu znajdowałby się bezpośrednio za literą `a`. Jednak jeśli umieścimy znak escape przed środkowym cudzysłowem `"a\"b"`, to Amber uwzględni go jako ciąg znaków. Wówczas cały tekst zakończy się zaraz za literą `b`. Sposób ten jest powszechnie nazywany escape'owaniem.

Bash został napisany w języku C, gdzie końce ciągów znaków są oznaczone specjalnym znakiem zerowym. Jednakże z punktu widzenia pisania skryptu Bash, użytkownik nie ma dostępu do tej informacji. Amber jest transpilowany do języka Bash, zatem przyjmuje on również taką samą strategię i pozwala powłoce zająć się tym problemem.

Kodowanie znaków w języku Bash i Amber uzależnione jest od ustawień środowiskowych systemu. Zazwyczaj jest to UTF-8, ale możliwe jest ustawienie dowolnego innego kodowania wspieranego przez system.

3.1.2. Liczba

Liczba w języku Amber reprezentowana jest poprzez typ `Num`. Bash przechowuje liczby wewnętrznie w tablicy znaków `char*` w reprezentacji dziesiętnej. Tak naprawdę nie ma różnicy między liczbami oraz ciągiem znaków w środowisku Bash. Liczby w języku Amber są transpilowane do Bash, co oznacza, że możemy przechowywać liczby z dowolną precyzją kosztem dużego zużycia pamięci. Nie jest to jednak zła wiadomość, ponieważ Bash nie został stworzony do wykonywania skomplikowanych obliczeń. W dalszych rozdziałach dowiemy się więcej na temat utraty precyzji podczas wykonywania operacji arytmetycznych. Aktualnie Amber wspiera jedynie zapis liczb dziesiętnych.

```
// Typ danych `Num`
// Może być liczbą całkowitą
42
// lub liczbą zmiennoprzecinkową
-123.456
```

3.1.3. Boolean

Wartości boolowskie są transpilowane do wartości numerycznych. Po transpilacji wartość `false` zamienia się w `0`, a wartość `true` zamienia się w `1`. Ponieważ Bash przechowuje wartości numeryczne tak samo jak wartości tekstowe, każda wartość boolowska zajmuje dwa bajty w pamięci (jeden bajt na samą wartość oraz drugi na znak końca ciągu znaków).

```
// Typ danych `Bool`
true
false
```

3.1.4. Tablica

Tablice w języku Amber jak i Bash są dynamicznie alokowane. W pliku źródłowym projektu Bash `array.h` [5], struktura tablicy z języka Bash przechowuje takie

informacje jak liczbę, oraz tablicę elementów, gdzie każdy z nich jest definiowany jako struktura zawierająca wartość w postaci ciągu znaków `char*` oraz jego indeks w tablicy.

```
typedef struct array {
    arrayind_t    max_index;
    arrayind_t    num_elements;
#ifdef ALT_ARRAY_IMPLEMENTATION
    arrayind_t    first_index;
    arrayind_t    alloc_size;
    struct array_element **elements;
#else
    struct array_element *head;
    struct array_element *lastref;
#endif
} ARRAY;

typedef struct array_element {
    arrayind_t    ind;
    char    *value;
#ifndef ALT_ARRAY_IMPLEMENTATION
    struct array_element *next, *prev;
#endif
} ARRAY_ELEMENT;
```

Istnieje też alternatywna implementacja tablicy w języku Bash, która działa jako lista wiązana. Dla takiej implementacji każdy element w liście przechowuje również wskaźnik do następnego jak i poprzedniego elementu.

Tworząc tablicę w języku Amber, tworzymy jej odpowiednik w języku Bash. Amber jest typowany, zatem wymaga przechowywania jednego ustalonego typu w danej tablicy. W przypadku gdy jest pusta, wymagane jest podanie typu wewnątrz nawiasów. Nie można umieszczać elementów różnych typów w tej samej tablicy.

```
// Typ danych `[Num]`
[1, 2, 3]
// Pusta tablica `[Text]`
[Text]
```

Z powodu wcześniej wspomnianych ograniczeń języka Bash implementacja tablic wielowymiarowych jest dość trudna, ponieważ wartości tablic są przechowywane jako

ciągi znaków. Na chwilę obecną Amber nie obsługuje zagnieżdżonych tablic.

```
[[Bool]]
// Error: Arrays cannot be nested due to the Bash limitations
```

3.2. Wyrażenia

Operatory umożliwiają działanie tylko na **tym samym** rodzaju danych. Na przykład dodanie `Text` do `Num` spowoduje błąd i nie jest obsługiwane.

3.2.1. Operator dodawania

Dodawanie może być wykonywane na liczbach, tekście i tablicach. Operator ten stosowany na różnych typach danych zwraca różne rezultaty:

- `Num` - suma arytmetyczna.
- `Text` - łączenie (konkatenacja) ciągów znaków.
- `[]` - łączenie (konkatenacja) tablic.

```
12 + 42 // 54
"Hello " + "World!" // "Hello World!"
[1, 2] + [3, 4] // [1, 2, 3, 4]
```

3.2.2. Operacje arytmetyczne

Operacje arytmetyczne mogą być używane tylko na typie danych `Num`. Oto lista wszystkich dostępnych operatorów:

- `+` Suma arytmetyczna
- `-` Odejmowanie
- `*` Mnożenie
- `/` Dzielenie
- `%` Operacja modulo

Wewnętrznie kompilator wykorzystuje komendę `bc`. Jest to kalkulator języka arytmetyki dokładnej z precyzją dziesiętną. Amber stosuje precyzję do 20 liczb po przecinku dla operacji arytmetycznych. Wyniki każdej operacji będą obcięte do 20 znaku po przecinku, ponieważ `bc` nie stosuje zaokrąglenia.

3.2.3. Komparatory

Operatory równości `==` i nierówności `!=` mogą być stosowane na wszystkich typach pod warunkiem, że porównujemy ten sam rodzaj danych.

Inne operacje porównania mogą być tylko używane na typie danych `Num`. Działają one tak samo jak w innych językach programowania: `>`, `<`, `>=`, `<=`. Wyrażenie operacji porównania zawsze ewaluuje się do typu `Bool`.

```
// Poprawne porównanie, które ewaluuje się do fałszu
"hello" == "world"
// Niepoprawne porównanie. Brak zgodności typów danych
"hello" != 42
// Niepoprawne porównanie. Lewa strona jest typem danych `Text`
"Lorem ipsum" > 5
```

3.2.4. Operacje logiczne

Operacje logiczne mogą być używane tylko na typie danych `Bool` i ich wyrażenia ewaluują się do typu `Bool`. W przeciwieństwie do rodziny języków programowania podobnych do C, zdecydowałem się na bardziej Python'owe podejście z słowami kluczowymi (`and`, `or`, `not`) zamiast symboli (`&&`, `||`, `!`), ponieważ lepiej pasują one do natury języka programowania skryptowego.

```
18 >= 12 and not false
```

3.2.5. Operator skrócony

Można używać operatora dodawania, oraz innych operatorów arytmetycznych połączonych z symbolem `=`, aby automatycznie przypisać wynik do istniejącej zmiennej:

```
let age = 18
age += 5
echo age // Wypisuje: 23
```

3.2.6. Interpolacja tekstu

Interpolacja wyrażeń w tekście jest to forma umieszczania w nim wartości różnych danych, które ostatecznie zostaną zamienione na ich reprezentację tekstową. Wyrażenie interpolacji rozpoczyna znak `{` oraz kończy go `}`.

```
echo "Stan: {false}" // Wypisuje: "Stan: 0"
// Możliwe jest również zagnieżdżanie interpolacji
echo "1 {" 2 {"3"} 4"} 5" // Wypisuje: "1 2 3 4 5"
```

W poniższej tabelce możemy zaobserwować w jaki sposób ona zachodzi dla różnych rodzajów danych:

Typ	Opis	Przed	Po
`Text`	Identyczność	`"{"Tekst"}"`	`"Tekst"`
`Num`	Identyczność	`"{12.34}"`	`"12.34"`
`Bool`	Zamiana na `1` lub `0`	`"{true}"`	`"1"`
`[]`	Umieszcza spacje między elementami	`"{[1, 2, 3]}"`	`"1 2 3"`

```
let name = "Damian"
let age = 18
echo "Cześć, jestem {name}. Mam {age} lat."
// Wypisuje: Cześć, jestem Damian. Mam 18 lat.
```

3.3. Zmienne

Zmienne są „mutowalne”, co oznacza, że można modyfikować ich stan w dowolnym momencie. Aby utworzyć zmienną, używamy słowa kluczowego `let`. Oto przykład:

```
let name = "Jan"
```

Jeśli zmienna już istnieje, możemy nadpisać jej wartość używając jej nazwy (bez użycia słowa kluczowego `let`):

```
name = "Robert"
```

Aby uzyskać dostęp do wartości przechowywanej przez tą zmienną, wystarczy odwołać się do niej po nazwie, w ten sposób:

```
echo name // Wypisuje: "Robert"
```

W pliku źródłowym powłoki Bash `variables.h` [6] możemy zauważyć, że zmienna jest przechowywana w pamięci jako struktura zawierająca między innymi informację o nazwie zmiennej oraz jej wartość w formie ciągu znaków `char*`.

```
typedef struct variable {
  /* Symbol that the user types. */
  char *name;
  /* Value that is returned. */
  char *value;
  /* String for the environment. */
  char *exportstr;
  /* Function called to return a `dynamic'
     value for a variable, like $SECONDS
     or $RANDOM. */
  sh_var_value_func_t *dynamic_value;
  /* Function called when this `special
     variable' is assigned a value in
     bind_variable. */
  sh_var_assign_func_t *assign_func;
  /* export, readonly, array, invisible... */
  int attributes;
  /* Which context this variable belongs to. */
  int context;
} SHELL_VAR;
```

3.3.1. Przesłanianie

Deklaracje zmiennych mogą być przesłaniane. Oznacza to, że możemy ponownie zadeklarować istniejącą zmienną z innym typem danych w danym zasięgu, jeśli jest to konieczne. Oto przykład:

```
// `result` jest `Num`
let result = 123
// `result` jest `Text`
let result = "Ala ma kota."
```

3.4. Warunki

Istnieją trzy sposoby realizacji logiki warunkowej.

3.4.1. Instrukcja If

Instrukcja, która w zależności od warunku wykonuje odpowiednią gałąź przepływu danych. Wyrażenie warunkowe musi być typem `Bool`. Jeśli jest prawdą, to wykona się blok dla przypadku prawdziwego, jeśli jest fałszem, to wykona się blok dla przypadku nieprawdziwego `else` (o ile istnieje).

```
if number % 2 == 0 {
    echo "Liczba jest parzysta"
}
```

Zamiast bloku kodu możemy umieścić jedno wyrażenie tuż po symbolu dwukropka, co pozwala skrócić zapis tej instrukcji jeszcze bardziej.

```
if number % 2 == 0: echo "Parzysta"
else: echo "Nieparzysta"

// Lub

if number % 2 == 0:
    echo "Parzysta"
else:
    echo "Nieparzysta"
```

3.4.2. Łańcuch If

Łańcuch if, to instrukcja, która transpilowana jest do sekwencji instrukcji if-else w języku Bash. Użycie jej pomaga w utrzymaniu czytelności kodu gdy wiele

instrukcji warunkowych musi być wykonywanych po sobie. W zapisie łańcucha if pomijamy wyrażenie warunkowe umieszczone bezpośrednio po słowie kluczowym `if` i umieszczamy je w gałęziach wewnątrz struktury. Każdy warunek będzie ewaluowany jeden po drugim dopóki nie będzie on prawdą. Wówczas wykona się jedynie blok tego warunku.

```
if {
    drink == "water" {
        echo "Zakupiono wodę"
    }
    drink == "cola" {
        echo "Zakupiono colę"
    }
    else {
        echo "Przepraszamy, nie mamy tego produktu"
    }
}

// Alternatywnie, jak wcześniej wspomniano:

if {
    drink == "water": echo "Zakupiono wodę"
    drink == "cola": echo "Zakupiono colę"
    else: echo "Przepraszamy, nie mamy tego produktu"
}
```

Równoważna struktura łańcucha if zapisana za pomocą struktury if-else:

```
if drink == "water" {
    echo "Zakupiono wodę"
} else {
    if drink == "cola" {
            echo "Zakupiono colę"
    } else {
            echo "Przepraszamy, nie mamy tego produktu"
    }
}
```

To podejście zapewnia bardziej zwięzłą i czytelniejszą strukturę do obsługi wielu warunków.

3.4.3. Wyrażenie trójargumentowe

Wyrażenia trójargumentowe pomagają nam zwięźle zwrócić wartość na podstawie warunku.

```
condition
    then branch_true_expression
    else branch_false_expression
```

Wyrażenie warunkowe poprzedzone jest słowem kluczowym `then`, po którym istnieje gałąź z wyrażeniem zwróconym w przypadku gdy warunek jest prawdą. Następnie `else`, po którym jest gałąź z wyrażeniem zwróconym w przypadku gdy warunek jest fałszem.

```
let candy = count > 1
    then "cukierki"
    else "cukierek"

echo "Mam {count} {candy}"
```

Wyrażenie trójargumentowe może być oczywiście użyte w jednym wierszu.

```
let candy = count > 1 then "cukierki" else "cukierek"
```

3.5. Polecenia

Jedynym sposobem dostępu do powłoki Bash w języku Amber jest użycie poleceń. Są to skrawki kodu Bash umieszczone pomiędzy znakami $. Polecenia mogą być używane jako instrukcje lub wyrażenia, przykład:

```
// Wypisuje zawartość pliku `file.txt`
unsafe $cat file.txt$
// Przypisuje zawartość pliku do zmiennej
let result = unsafe $cat file.txt$
```

Wynik instrukcji polecenia jest zapisywany do standardowego wyjścia, natomiast wynik wyrażenia polecenia jest przypisywany do zmiennej w typie `Text`.

Polecenia są instrukcjami i wyrażeniami *zawodnymi*. To oznacza, że wymagają obsłużenia potencjalnego błędu. W przypadku powyższej składni korzystamy ze słowa kluczowego `unsafe`, które informuje kompilator o zignorowaniu tego przypadku. Nie jest to zalecane, gdyż utrudnia diagnozowanie błędów i pozwala na dalsze wykonywanie kodu jak w scenariuszu sukcesu. Istnieje kilka sposobów obsługi błędu:

- `failed` - pozwala na wykonanie dodatkowych instrukcji w przypadku niepowodzenia polecenia.
- `?` - upraszcza propagację błędów, automatycznie zwracając wartość błędu z funkcji.
- `unsafe` - odradzany sposób obsługi błędów. To modyfikator polecenia, który ignoruje możliwość wystąpienia błędu.

3.5.1. Podstawowa obsługa błędów

Standardowa obsługa błędów `failed` wymaga zadeklarowania bloku, który ma się wykonać w przypadku niepowodzenia, przykład:

```
// Instrukcja polecenia
$mv file.txt dest.txt$ failed {
    echo "Nie można otworzyć pliku"
}

// Wyrażenie polecenia
let result = $cat file.txt | grep "GOTOWY"$ failed {
    echo "Nie można otworzyć pliku"
}
echo result
```

3.5.2. Propagacja błędów

Podczas wywoływania polecenia wewnątrz funkcji, obsługę błędu można przekazać do kontekstu, z którego została wywołana. Wówczas funkcja ta automatycznie staje się *zawodną*.

```
fun foo() {
    $test -d /ścieżka/do/pliku$?
}
```

```
foo() failed {
    echo "Wywołanie funkcji się nie powiodło"
}
```

3.5.3. Interpolacja wyrażeń

Polecenie może być również interpolowane z innymi wyrażeniami. Proces ten przebiega dokładnie tak samo jak w typie danych `Text`.

```
let filePath = "/ścieżka/do/pliku"
$cat {filePath}$ failed {
    echo "Nie można otworzyć {filePath}"
}
```

3.5.4. Uzyskiwanie kodu wyjścia

Aby uzyskać kod wyjścia ostatnio wykonanego polecenia, możemy użyć słowa kluczowego `status`.

```
let filePath = "/ścieżka/do/pliku"
$cat {filePath}$ failed {
    echo "Błąd! Kod wyjścia: {status}"
}
echo "Kod statusu to: {status}"
```

3.6. Modyfikatory poleceń

Poznaliśmy wcześniej modyfikator `unsafe`. Poza nim istnieje też `silent`, który przekierowuje wyjście polecenia do `/dev/null`, skutecznie je wyciszając. Możemy użyć więcej niż jeden modyfikator dla pojedynczego polecenia:

```
silent unsafe $tar -czvf archive.tar.gz /path/to/directory$
```

Podobnie, możemy definiować bloki kodu obejmujące kilka poleceń, stosując modyfikatory zakresu. Tworzą one kontekst w którym modyfikatory będą automatycznie przypisywane wszystkim komendom jak i funkcjom zawodnym wywołanym wewnątrz. Oto przykład zastosowania modyfikatorów zakresu:

```
unsafe silent {
    $git pull origin main$
    $systemctl restart my_application.service$
}
```

W powyższym przykładzie wszystkie polecenia są wykonywane w kontekście modyfikatorów `unsafe` i `silent`, co oznacza, że są one wykonywane bez dodatkowych zabezpieczeń oraz ich wyjście jest wyciszone.

3.7. Tablice

Indeksowanie tablic rozpoczyna się od zera.

```
let groceries = ["Ananas", "Banan", "Cytryna"]
groceries[0] = "Arbuz"
echo groceries[1]
// Wyjście: Banan
echo groceries
// Wyjście: Arbuz Banan Cytryna
```

Dodawanie nowego elementu do tablicy może być obsłużone przez konkatenację tablic.

```
let capitals = ["Londyn", "Paryż"]
capitals += ["Warszawa"]

let cities = capitals + ["Barcelona", "Wrocław"]
```

Usuwanie elementów z tablicy nie zostało jeszcze zaimplementowane. Zamierzam to zrobić w dalszym rozwoju projektu. Może to zostać obsłużone poprzez polecenie Bash.

3.8. Zakresy

Amber umożliwia generowanie tablicy liczb całkowitych `[Num]` z określonego przedziału:

- `a..b` lewostronnie domknięty $\langle a, b)$.

- `a..=b` obustronnie domknięty $\langle a, b \rangle$.

Przykład:

```
echo 0..10
// Wyjście: 0 1 2 3 4 5 6 7 8 9

echo 0..=10
// Wyjście: 0 1 2 3 4 5 6 7 8 9 10
```

3.9. Pętle

Amber obsługuje dwa typy pętli:

- **Nieskończona**, która może być przerwana tylko za pomocą słowa kluczowego `break`.
- **Pętla iteratora**, która iteruje po elementach tablicy.

W kontekście pętli możemy używać słów kluczowych `break` oraz `continue`, aby lepiej kontrolować przepływ logiki.

3.9.1. Nieskończona pętla

```
let i = 0
let sum = 0
loop {
    i += 1
    if i == 5: break
    sum += i
}
echo sum
// Wyjście: 10
```

3.9.2. Pętla iteratora

Przykład działania pętli iteratora po zakresie:

```
let sum = 0
loop i in 0..5 {
    sum += i
}
echo sum
// Wyjście: 10
```

Pętle iteratora mogą również zapewniać dostęp do indeksu w bieżącej iteracji.

```
let files = ["config.json", "file.txt", "audio.mp3"]

loop index, file in files {
    $mv {file} {index}{file}$ failed {
        echo "Nie udało się zmienić nazwy {file}"
    }
}
```

3.10. Funkcje

Funkcje umożliwiają zgrupowanie zestawu instrukcji w celu realizacji określonego zadania. Deklaracja funkcji rozpoczyna się od słowa kluczowego `fun`. Istnieją dwie metody deklarowania funkcji: z ściśle określonymi typami oraz bez nich co umożliwia większą elastyczność w definiowaniu funkcji.

```
fun strictFunction(arg1: Num, arg2: Num): Num {
    let result = arg1 + arg2
    return result
}

echo strictFunction(2, 3)
// Wyjście: 5
echo strictFunction("Cześć", " Świecie")
// Błąd: pierwszy argument 'arg1' funkcji 'foo'
// oczekuje typ 'Num', ale napotkał 'Text'

fun genericFunction(arg1, arg2) {
    let result = arg1 + arg2
    return result
}
```

```
echo myFunction(2, 3)
// Wyjście: 5
echo myFunction("Cześć", " Świecie")
// Wyjście: Cześć Świecie
```

Funkcje nie są analizowane składniowo do momentu ich użycia. W momencie wywołania funkcji, Amber generuje wariant tej funkcji w zależności od typów argumentów, które są do niej przekazywane.

3.10.1. Obsługa błędów

Funkcje mogą zawieść, jeśli wykonają instrukcję `fail` zawierającą kod wyjścia. Takie funkcje określane są mianem *zawodnych*. Przykłady takich funkcji:

```
fun foo() {
    fail 1
}

fun bar(name) {
    $command$?
    parse(name)?
}
```

Możliwe jest stosowanie operatora propagacji obsługi błędów tak samo jak w przypadku poleceń. Kod wyjścia dla funkcji zawodnych jest również dostępny za pomocą słowa kluczowego `status`.

3.11. Importowanie

W języku Amber istnieje możliwość importowania funkcji z innych plików. Aby funkcja była dostępna z zewnętrznego pliku, konieczne jest uprzednie zadeklarowanie jej jako *publicznej*.

3.11.1. Publiczne funkcje

Funkcję publiczną definiuje słowo kluczowe `pub` poprzedzone słowem `fun`.

```
pub fun sum(left: Num, right: Num): Num {
    return left + right
}
```

3.11.2. Importowanie z innych plików

Importowanie funkcji wymaga podania ścieżki do pliku.

```
import { foo, bar as car } from "./my-file.ab"
import * from "./arith.ab"

foo()
car()
echo sum(1, sub(2, mul(4, 5)))
```

3.11.3. Publiczne importy

Istnieje również możliwość automatycznego upublicznienia funkcji, które zostały zaimportowane z innego pliku.

```
pub import { foo, bar } from "path/to/file.ab"
```

3.11.4. Blok main

Zdarza się, że zachodzi potrzeba wykonania określonego fragmentu kodu w przypadku, gdy plik jest uruchamiany bezpośrednio, a nie jest importowany jako zależność. W Pythonie ten problem można rozwiązać następująco:

```
if __name__ == '__main__':
    # Kod do wykonania
```

Amber zapewnia specjalną składnię `main` dla tego wzorca. Wewnątrz tego bloku możemy korzystać z operatora propagacji obsługi błędów, który przekaże błąd do powłoki i zakończy działanie skryptu.

```
echo "Pośrednie uruchomienie"

main {
    echo "Bezpośrednie uruchomienie"
    $jakieś polecenie$?
}
```

Uruchomienie tego pliku bezpośrednio zwraca następujący rezultat:

```
Pośrednie uruchomienie
Bezpośrednie uruchomienie
```

Gdy uruchomimy inny plik w którym ten jest importowany, otrzymamy:

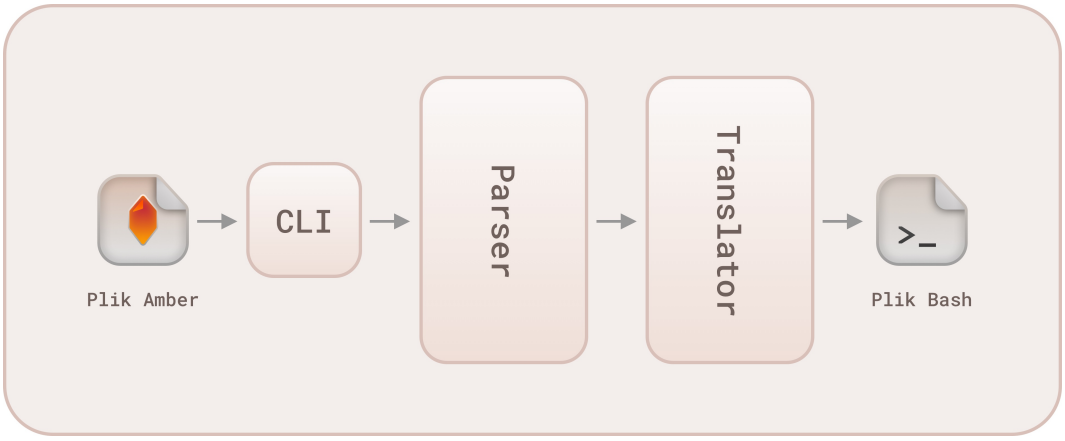
```
Pośrednie uruchomienie
```

Rozdział 4.

Ogólna budowa kompilatora

Amber jest napisany w języku Rust [1] i korzysta z framework'a Heraklit (również napisanego w tym języku), który będzie dokładniej opisany w kolejnych rozdziałach. Kompilator wykorzystuje interfejs wiersza poleceń (CLI) do interpretacji instrukcji wykonania. Następnie proces kompilacji zachodzi w dwóch fazach.

1. Pierwsza faza to **Parsowanie**.
2. Druga faza to **Translacja**.



Rysunek 4.1: Fazy kompilacji języka Amber.

4.1. Parsowanie

Lekser, który jest wbudowany w framework Heraklit, wymaga pierwotnie pewnej konfiguracji. Przyjmuje on na wejściu tak zwane zasady analizy leksykalnej, które definiują ogólne zasady tego jaki wektor tokenów ma wyprodukować.

4.1.1. Zasady Leksykalne

Zasady leksykalne zawierają się w trzech sekcjach:

1. **Symbole** - pojedyncze symbole, które powinny być traktowane jako osobne tokeny. W przypadku języka Amber są to między innymi: `(`, `)`, `+`, `=` etc.
2. **Zlepki** - zlepki symboli, które również powinny być traktowane jako osobne tokeny. Przykładowo `+=` jest zlepkiem, który składa się z symboli `+` oraz `=`.
3. **Regiony** - mogą mieć inne zasady parsowania wewnątrz. Definiowany jest poprzez znacznik rozpoczynający oraz znacznik zamykający. Przykładem regionu może być ciąg znaków `"Hello world"`.

Po otrzymaniu takich zasad leksykograficznych oraz wykonaniu etapu leksykacji Heraklit zwróci wektor tokenów, które zawierają informację leksykalną, czyli słowo, jak i oryginalną pozycję w pliku.

W przypadku gdy zajdzie nieoczekiwane zachowanie podczas parsowania, zwracany jest odpowiedni błąd. Przykładowo gdy kod, który zostanie dostarczony do leksera zawiera niedomknięty region, Heraklit zwraca błąd wykrycia niedomkniętego regionu, który można następnie wyświetlić na standardowym wyjściu.

4.2. Parsowanie tokenów

Aby efektywnie parsować ciąg tokenów, zalecane jest wykorzystanie stanu, który pozwala śledzić przykładowo użycia nowych zmiennych oraz zakresy w jakich są deklarowane. Aby temu pomóc, Heraklit implementuje minimalną wersję takiego stanu, który zawiera podstawowe informacje takie jak zbiór wszystkich tokenów oraz licznik aktualnie parsowanego tokenu. Ten typ danych również implementuje kilka funkcji pomocniczych, które zwracają aktualnie parsowany token konsumując go poprzez inkrementację wewnętrznego licznika.

4.2.1. Składnia EBNF języka Amber

Extended Backus-Naur Form (EBNF) to formalny sposób opisywania gramatyk języków programowania, który umożliwia precyzyjne definiowanie składni. Poniżej znajduje się opis języka Amber w tym formacie. Korzeniem drzewa składni jest węzeł `root`. Prawie wszystkie inne produkcje (opisane gramatyką bezkontekstową), o ile nie są dość trywialne, są zaprogramowane w formie modułów składniowych języka Amber, które znajdziemy w ścieżce `/src/modules`.

```
root = { statement_global } ;

(* Statement *)
statement_local = ( expression | variable_init | variable_set
    | loop | loop_array | if_statement | if_chain ) { "\n" } ;
statement_global = ( statement_local | function_def | main
    | import_all | import_ids ) { "\n" } ;

(* Block *)
singleline_block = ':', statement_local ;
multiline_block = '{', { statement_local }, '}' ;
block = singleline_block | multiline_block ;

(* Expression *)
expression = number | text | boolean | null | list | command
    | binary_operation | unary_operation | parentheses | ternary
    | range | range_inclusive | identifier | function_call ;
```

W powyższym zapisie tłumaczyć będziemy instrukcje jako *statement* oraz wyrażenia jako *expression*.

Zauważmy, że w języku istnieje zróżnicowanie między globalną i lokalną instrukcją. Ponadto, globalne instrukcje są wspominane jedynie w zakresie produkcji `root`, co oznacza, że nie możemy definiować nowych funkcji wewnątrz innych funkcji.

```
(* Terminals *)
ANY_CHAR = ? any character ? ;
LETTER = ? A..Z and a-z ? ;
DIGIT = ? 0-9 ? ;
TYPE = 'Text' | 'Num' | 'Bool' | 'Null' ;
UNARY_OP = 'not';
BINARY_OP = '+' | '-' | '*' | '/' | '%' | 'and'
    | 'or' | '==' | '!=' | '<' | '<=' | '>' | '>=' ;
COMMAND_MOD = 'silent' | 'unsafe' ;
VISIBILITY = 'pub' ;

(* Identifier *)
identifier = ( LETTER | '_' ) , { LETTER | '_' | DIGIT } ;
```

Identyfikatory oznaczone jako *identifier* w języku Amber są wykorzystywane do nazywania funkcji i zmiennych. W większości języków, również w języku Amber,

identyfikatory mogą rozpoczynać się od litery lub podkreślenia a następnie zawierać symbole alfanumeryczne oraz podkreślenia.

```
(* `Num` literal *)
integer = DIGIT , { DIGIT } ;
real = integer , '.' , integer ;
number = integer | real ;

(* `Text` literal *)
interpolation = '{' , expression , '}' ;
text = '"' , { ANY_CHAR | interpolation } , '"' ;

(* `Bool` literal *)
boolean = 'true' | 'false' ;

(* `Null` literal *)
null = 'null' ;

(* `List` literal *)
empty_list = '[' , TYPE , ']' ;
full_list = '[' , [ expression , { ',' , expression } ] , ']' ;
list = empty_list | full_list ;
```

Powyżej widzimy w jaki sposób konstruowane są poszczególne literały. Każdy literał przedstawia inny rodzaj danych. Zauważmy, że pusta tablica musi zawierać typ danych, którego dotyczy.

```
(* Command expression *)
(* The ordering of command modifiers doesn't matter *)
command_modifier = SILENT_MOD, [ UNSAFE_MOD ] ;
command_modifier_block = command_modifier, multiline_block ;
commnad_base = '$', { ANY_CHAR | interpolation }, '$' ;
command = [ SILENT_MOD ], commnad_base, [ failure_handler ] ;
command_unsafe = [ SILENT_MOD ], UNSAFE_MOD, commnad_base ;

(* Operations *)
binary_operation = expression , BINARY_OP , expression ;
unary_operation = UNARY_OP , expression ;

(* Parentheses *)
parentheses = '(', expression, ')' ;
```

```
(* Failure handler *)
failure_propagation = '?' ;
failure_block = 'failed', block ;
failure_handler = failure_propagation | failure_block ;
```

Jako, że składnia EBNF nie pozwala skrótowo zapisać dowolnej permutacji zależności składniowych, oznaczone jest komentarzem, że ciąg modyfikatorów jest poprawny w dowolnej permutacji. Niepoprawne jest jednak powtarzanie pojedynczego modyfikatora.

```
(* Variable *)
variable_index = '[', expression, ']' ;
variable_init = 'let', identifier, '=', expression ;
variable_get = identifier, [ variable_index ] ;
variable_set = identifier, [ variable_index ], '=', expression ;

(* Function *)
function_call_params = [ expression, { ',', expression } ] ;
function_call = [ SILENT_MOD ],
    identifier, '(', function_call_params, ')' ;

function_call_failed = function_call, failure_handler ;
function_call_unsafe = UNSAFE_MOD, function_call ;

function_def = [ VISIBILITY ], 'fun', identifier, '(',
    [ identifier, { ',', identifier } ],
    ')', block ;
function_def_typed = [ VISIBILITY ], 'fun', identifier, '(',
    [ identifier, ':', TYPE, { ',', identifier, ':', TYPE } ],
    ')', ':', TYPE, block ;
```

Powyżej widzimy, że możemy wyciszyć funkcję, która zapisuje do standardowego wyjścia. W ten sposób możemy umieścić informacje pomagające zdebugować funkcję, a następnie, gdy nie będą nam potrzebne, wyciszyć ją jednym modyfikatorem.

```
(* Loop *)
loop = 'loop', block ;
loop_array = 'loop', identifier, 'in', expression, block ;
loop_array_iterator = 'loop', identifier, ',', identifier,
```

```
    'in', expression, block ;

(* Ranges *)
range = expression, '..', expression ;
range_inclusive = expression, '..=', expression ;

(* Conditional *)
if_statement = 'if', expression, block, [ 'else', block ] ;
if_chain = 'if', '{',
    { expression, block }, [ 'else', block ],
    '}' ;
ternary = expression, 'then', expression, 'else', expression ;

(* Main *)
main = 'main', [ '(', identifier, ')' ], block ;

(* Imports *)
import_path = '"', { ANY_CHAR }, '"' ;
import_all = [ VISIBILITY ], 'import', '*', 'from', import_path ;
import_ids = [ VISIBILITY ], 'import', '{',
    { identifier, [ 'as', identifier ] },
    '}', 'from', import_path ;

(* Comment *)
comment = '//', { ANY_CHAR }, '\n' ;
```

Aktualnie komentarze blokowe nie są wspierane, ponieważ nie jest to kluczowym elementem projektu. Wsparcie dla nich planuję dodać w przyszłości.

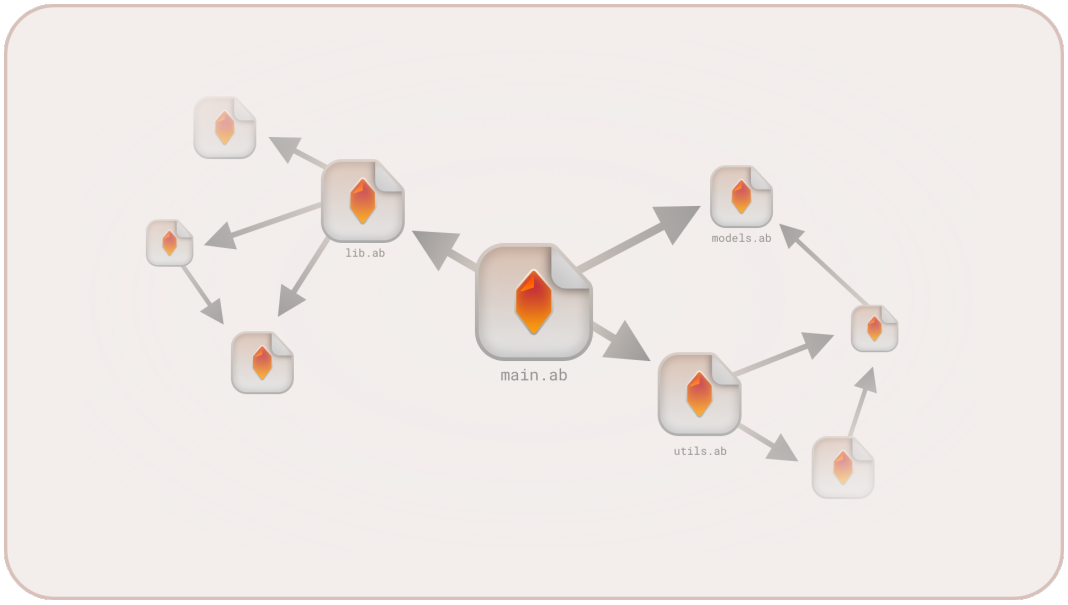
4.3. Algorytmy użyte w kompilatorze

Pomimo tego, że sam parser jest pewnym rodzajem algorytmu DFS, to istnieje wiele innych ciekawych algorytmów, które wykorzystałem podczas budowy języka Amber.

4.3.1. Importowanie plików

Importy plików tworzą graf, który nie może zawierać cykli. Z tego powodu aby zaimplementować poprawne importowanie plików sortuję je pod kątem zależności.

Jeśli plik `main.ab` importuje plik `lib.ab`, to w ostatecznym pliku Bash zawartość pliku `main.ab` musi być poprzedzona zawartością pliku `lib.ab`.



Rysunek 4.2: Przykładowy graf zależności plików źródłowych języka Amber.

Aby uzyskać poprawny graf zależności, musimy posortować zależności w sposób rosnący po stopniu wejściowym wierzchołka. Wykorzystuję do tego standardowy algorytm sortowania topologicznego [12], o złożoności $O(V + E)$.

4.3.2. Znajdowanie podobnych elementów

W przypadkach gdy programista popełnia błąd i wpisuje nazwę zmiennej lub funkcji zawierającej literówkę, kompilator języka Amber jest wstanie zidentyfikować zmienną, którą programista miał na myśli i zasugerować jej poprawną nazwę. Aby uzyskać ten efekt użyłem algorytmu LCS (najdłuższego wspólnego podciągu) [4] w wariancie o złożoności pamięciowej $O(n)$.

Charakterystyczne dla tego wariantu jest to, że nie zwraca wspólnego podciągu, a jedynie jego długość, co jest wystarczające dla naszych celów. Chcemy wiedzieć, jak długi będzie wspólny podciąg istniejącego identyfikatora oraz identyfikatora z literówką. Przy dopasowaniu brane są pod uwagę zmienne lub funkcje widoczne we wspólnym zasięgu.

```
pub fn find_best_similarity(
    taregt: impl AsRef<str>,
    options: &[impl AsRef<str>]
) -> Option<(String, f64)>;
```

Funkcja `find_best_similarity` zwraca najlepsze dopasowanie dla podanych dwóch ciągów i zwraca krotkę - liczbę reprezentującą punkty procentowe najbardziej dopasowanego ciągu oraz nazwę tego ciągu. Jeśli funkcja zwróci wartość poniżej 75%, to zakładam, że błędna nazwa nie jest literówką istniejącej zmiennej.

W kolejnych iteracjach planuję zmianę tego algorytmu na algorytm wyszukiwania odległości edycyjnej (odległość Levenshteina), który może się lepiej nadawać do tych celów.

Rozdział 5.

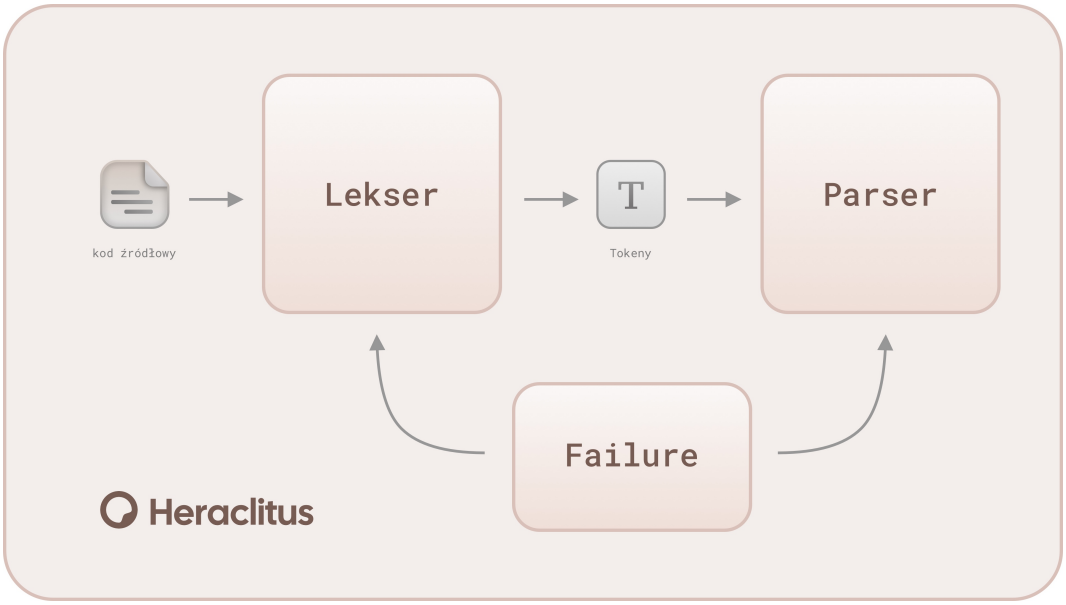
Heraklit

Heraklit (*Heraclitus*) to zaprojektowany przeze mnie framework do tworzenia języków programowania, które charakteryzują się takimi wspólnymi cechami:

- Składnia nie jest zależna od użycia ilości białych znaków w języku. Heraklit nie jest stworzony do leksowania i parsowania takich języków, jak Python, GDScript czy też Nim.
- Składnię można zapisać za pomocą:
  - **Słów** oddzielonych białymi znakami, które mogą być słowami kluczowymi lub identyfikatorami.
  - **Symboli**, czyli pojedynczych znaków, które same w sobie są osobnymi tokenami.
  - **Zlepków**, czyli symboli, które umieszczone obok siebie, tworzą jeden token.
  - **Regionów**, czyli ciągów znaków oznaczonych znacznikami rozpoczęcia i zakończenia.

Chociaż framework ma ograniczenia w tworzeniu składni, to jednak pozwala na parsowanie wielu popularnych języków programowania. Wspólny format języka programowania upraszcza reguły, które lekser musi stosować aby wygenerować tokeny. Wśród języków, które spełniają powyższe kryteria znajdują się między innymi:

- Język C
- JavaScript
- HTML lub też CSS



Rysunek 5.1: Opis budowy Heraklita.

5.1. Budowa Heraklita

Heraklit składa się trzech głównych komponentów:

- Lekser, który wykonuje analizę leksykalną języka, rozpoznając i klasyfikując tokeny.
- Parser, który tworzy strukturę metadanych (`Metadata`), która wspomaga parsowanie tokenów. Jest odpowiedzialny za analizę składniową języka i tworzenie drzewa parsowania.
- Failure, który zarządza błędami i umożliwia wypisanie ich przez kompilator. Wyróżniamy dwa rodzaje błędów:
  - **Głośny** - który jest najbardziej widoczny dla użytkownika, oznacza krytyczny błąd podczas analizowania składni.
  - **Cichy** - pojawia się wielokrotnie podczas kompilacji. Zwykle gdy struktura składniowa nie odpowiada wymaganiom modułu składniowego.

Rozdział 6.

Wtyczka do podświetlania składni języka Amber

Dla środowiska programistycznego Visual Studio Code zostało stworzone rozszerzenie, która podświetla składnię języka Amber. Wtyczki językowe tego środowiska obsługują gramatykę w formacie TextMate [14], która umożliwia tokenizację za pomocą wyrażeń regularnych. Dodatkowo format ten może być rozszerzany poprzez definiowanie bloków, które są podobne do regionów Heraklita. Definicja gramatyki w formacie TextMate jest zapisywana w języku JSON. Poniżej znajduje się fragment gramatyki języka Amber:

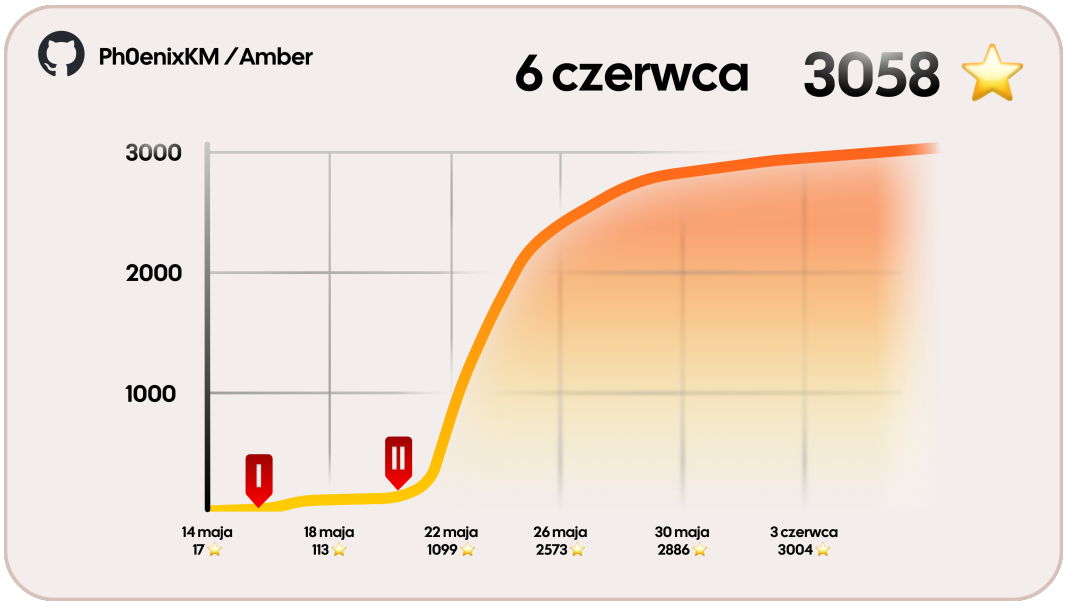
```
{
    "name": "keyword.control.amber",
    "match": "\\b(if|loop|ref|return|fun|else|then|break|...)\\b"
},
{
    "match": "\\b(true|false|null)\\b(?![?!])",
    "name": "constant.language.boolean.amber"
},
{
    "match": "[+-]?([0-9]*[.])?[0-9]+",
    "name": "constant.language.float.amber"
},
{
    "match": "\\b\\w+\\s*(?=\\([\\S\\s]*?\\)?)",
    "name": "entity.name.function"
},
```

Rozdział 7.

Premiera Produktu

Następnym krokiem było stworzenie kampanii marketingowej. Wybrałem następujące kanały promocji: Instagram, Hacker News, Reddit oraz X (dawniej znany jako Twitter).

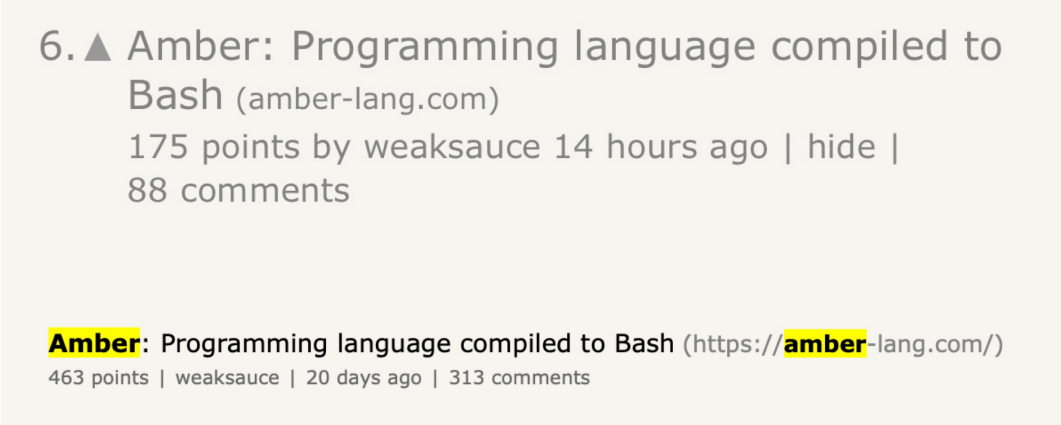
Kampania marketingowa rozpoczęła się 14 maja 2024 roku. W tym dniu utworzone zostało oficjalne konto języka Amber na Instagramie a także zostały zaprojektowane materiały promocyjne. Następnego dnia, 15 maja (oznaczonego czerwonym znacznikiem **I** na rysunku 7.1), opublikowane zostały pierwsze materiały reklamowe na portalach: Instagram, Hacker News, Reddit oraz X.



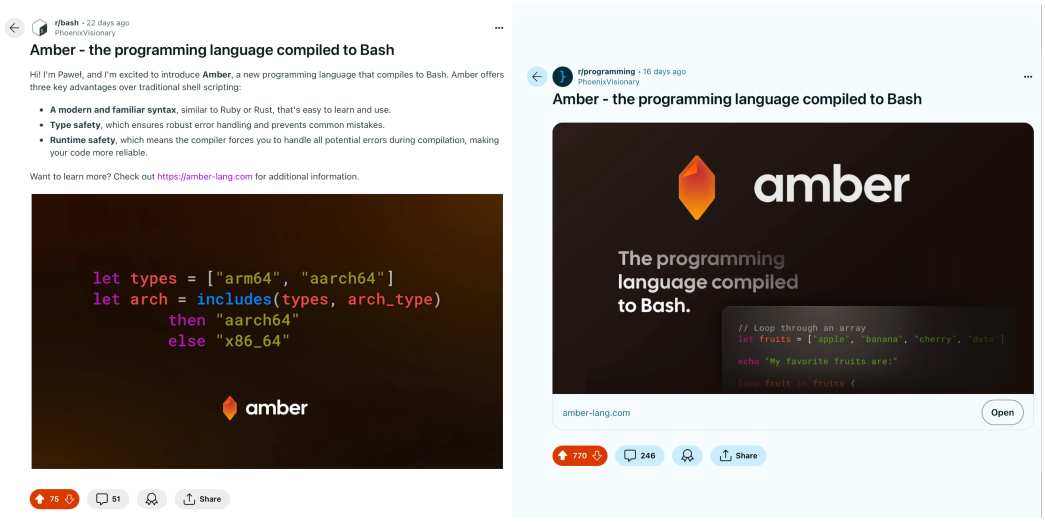
Rysunek 7.1: Liczba gwiazdek przyznanych repozytorium języka Amber na platformie GitHub w okresie trwania kampanii marketingowej.

Post Reddit, umieszczony na forum poświęconemu językowi Bash, zdobył popularność zyskując 75 głosów od użytkowników.

W pierwszym dniu publikacji post na Hacker News został oznaczony jako martwy (dead) i usunięty z forum - prawdopodobnie wskutek błędu moderatora. Tydzień później, 22 maja 2024 roku, otrzymałem email z informacją o przyznaniu artykułowi drugiej szansy - skorzystałem z tej okazji. Amber znalazł się na szóstym miejscu w rankingu najpopularniejszych wiadomości na Hacker News utrzymując swoją pozycję przez kilka dni. W tym czasie liczba gwiazdek w repozytorium (oznaczająca popularność projektu) wynosiła już 1099. Ostatecznie artykuł na Hacker News zdobył ponad 400 punktów.



Mniej więcej w tym samym czasie opublikowany został artykuł na forum programistycznym, na platformie Reddit. Forum to ma większą liczbę użytkowników i jest bardziej popularne, co przyczyniło się do większej aprobaty (zdobył 700 głosów). Wkrótce po tym zaczęły pojawiać się osoby, które zaczęły naprawiać błędy w kodzie i implementować nowe funkcjonalności.



Rysunek 7.2: Artykuły opublikowane na platformie Reddit.

Do dzisiaj repozytorium języka Amber współtworzyło aż piętnaście nowych osób. Utworzyłem także serwer Discord dla społeczności tego języka. Obecnie serwer liczy

ponad 100 użytkowników z czego 45 osób wybrało rolę *Contributor*, co oznacza, że planują lub już uczestniczyli w rozwoju języka.

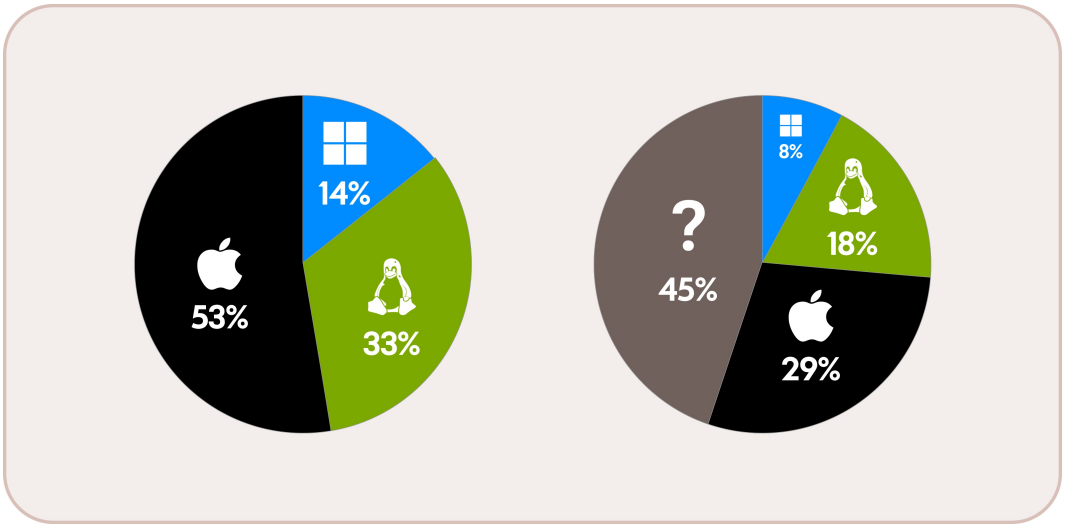
7.1. Statystyki zaangażowania

Na stronie internetowej jak i w instalatorze samego języka umieszczony został kod telemetryczny. Dzięki temu możliwe było zebranie statystyk dotyczących liczby wejść na stronę dokumentacji, repozytorium oraz liczby pobrań instalatora, a także zebranie danych identyfikujących system użytkowników.

Liczba wejść na stronę dokumentacji	41 352
Liczba wejść na stronę repozytorium	8 410
Całkowita liczba pobrań instalatora	2 266

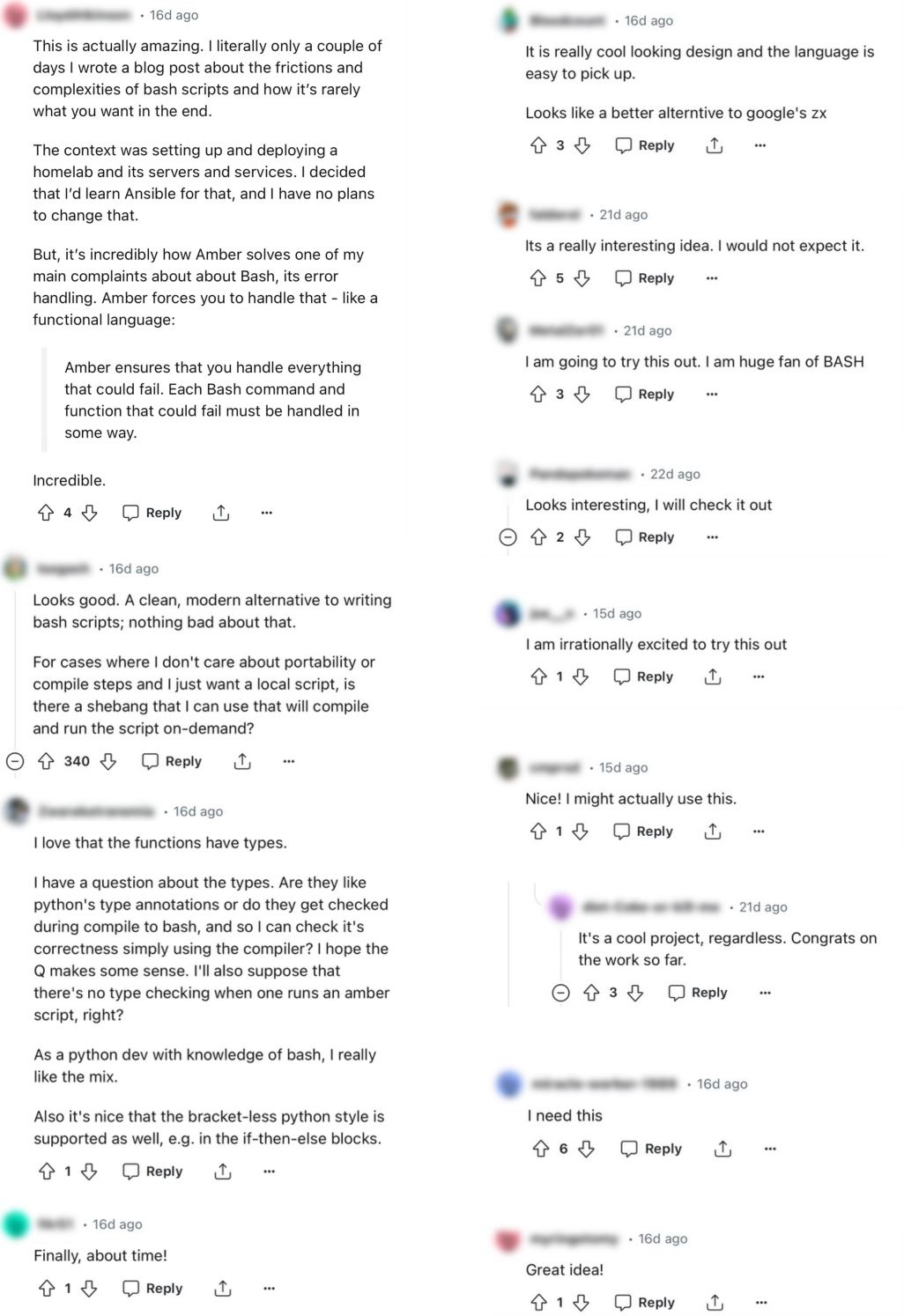
Tabela 7.1: Statystyki zebrane za pomocą telemetrycznego kodu z dnia 6 czerwca 2024.

Do klasyfikacji systemów operacyjnych użyto polecenia `uname`. To samo polecenie jest użyte w instalatorze do wykrycia systemu. Z całego zbioru danych tylko 55% instalacji zostało poprawnie sklasyfikowanych. Według mojej oceny 45% nieskategoryzowanych systemów operacyjnych prawdopodobnie to systemy Linux w dystrybucji, która nie ma zainstalowanego polecenia `uname`.



Rysunek 7.3: Wykres kołowy instalacji kompilatora Amber z uwzględnieniem i bez uwzględnienia niesklasyfikowanych systemów operacyjnych.

7.2. Oceny użytkowników na portalu Reddit



[redacted] • 16d ago

This is actually amazing. I literally only a couple of days I wrote a blog post about the frictions and complexities of bash scripts and how it's rarely what you want in the end.

The context was setting up and deploying a homelab and its servers and services. I decided that I'd learn Ansible for that, and I have no plans to change that.

But, it's incredibly how Amber solves one of my main complaints about about Bash, its error handling. Amber forces you to handle that - like a functional language:

> Amber ensures that you handle everything that could fail. Each Bash command and function that could fail must be handled in some way.

Incredible.

4 Reply

[redacted] • 16d ago

Looks good. A clean, modern alternative to writing bash scripts; nothing bad about that.

For cases where I don't care about portability or compile steps and I just want a local script, is there a shebang that I can use that will compile and run the script on-demand?

340 Reply

[redacted] • 16d ago

I love that the functions have types.

I have a question about the types. Are they like python's type annotations or do they get checked during compile to bash, and so I can check it's correctness simply using the compiler? I hope the Q makes some sense. I'll also suppose that there's no type checking when one runs an amber script, right?

As a python dev with knowledge of bash, I really like the mix.

Also it's nice that the bracket-less python style is supported as well, e.g. in the if-then-else blocks.

1 Reply

[redacted] • 16d ago

Finally, about time!

1 Reply

[redacted] • 16d ago

It is really cool looking design and the language is easy to pick up.

Looks like a better alterntive to google's zx

3 Reply

[redacted] • 21d ago

Its a really interesting idea. I would not expect it.

5 Reply

[redacted] • 21d ago

I am going to try this out. I am huge fan of BASH

3 Reply

[redacted] • 22d ago

Looks interesting, I will check it out

2 Reply

[redacted] • 15d ago

I am irrationally excited to try this out

1 Reply

[redacted] • 15d ago

Nice! I might actually use this.

1 Reply

[redacted] • 21d ago

It's a cool project, regardless. Congrats on the work so far.

3 Reply

[redacted] • 16d ago

I need this

6 Reply

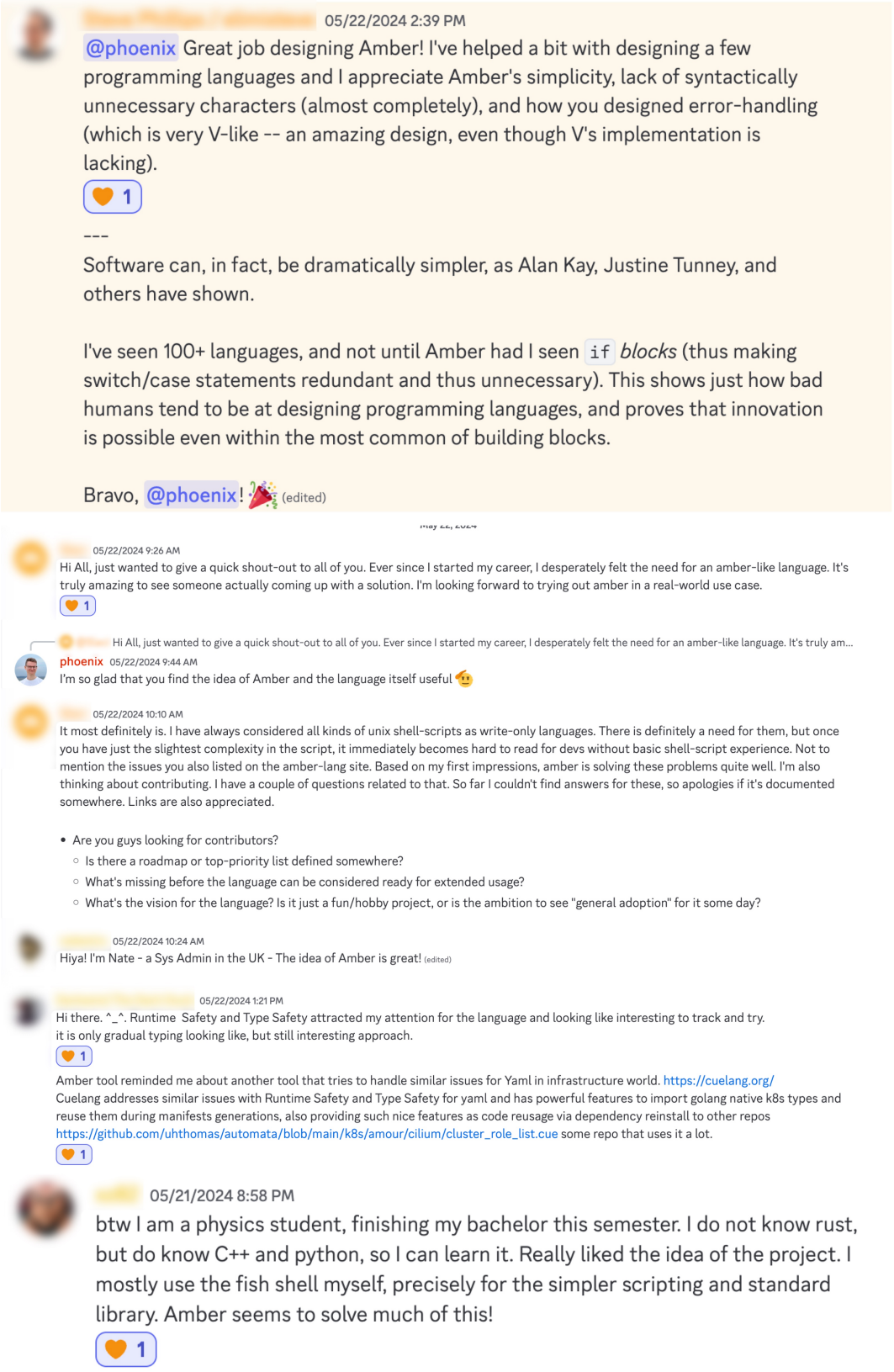
[redacted] • 16d ago

Great idea!

1 Reply

7.3. Posty użytkowników na portalu X

7.4. Informacja zwrotna członków serwera Discord



[redacted] 05/22/2024 2:39 PM

@phoenix Great job designing Amber! I've helped a bit with designing a few programming languages and I appreciate Amber's simplicity, lack of syntactically unnecessary characters (almost completely), and how you designed error-handling (which is very V-like -- an amazing design, even though V's implementation is lacking).

1

Software can, in fact, be dramatically simpler, as Alan Kay, Justine Tunney, and others have shown.

I've seen 100+ languages, and not until Amber had I seen `if` *blocks* (thus making switch/case statements redundant and thus unnecessary). This shows just how bad humans tend to be at designing programming languages, and proves that innovation is possible even within the most common of building blocks.

Bravo, @phoenix! (edited)

[redacted] 05/22/2024 9:26 AM

Hi All, just wanted to give a quick shout-out to all of you. Ever since I started my career, I desperately felt the need for an amber-like language. It's truly amazing to see someone actually coming up with a solution. I'm looking forward to trying out amber in a real-world use case.

1

[redacted] Hi All, just wanted to give a quick shout-out to all of you. Ever since I started my career, I desperately felt the need for an amber-like language. It's truly am...

phoenix 05/22/2024 9:44 AM

I'm so glad that you find the idea of Amber and the language itself useful

[redacted] 05/22/2024 10:10 AM

It most definitely is. I have always considered all kinds of unix shell-scripts as write-only languages. There is definitely a need for them, but once you have just the slightest complexity in the script, it immediately becomes hard to read for devs without basic shell-script experience. Not to mention the issues you also listed on the amber-lang site. Based on my first impressions, amber is solving these problems quite well. I'm also thinking about contributing. I have a couple of questions related to that. So far I couldn't find answers for these, so apologies if it's documented somewhere. Links are also appreciated.

- Are you guys looking for contributors?
  - Is there a roadmap or top-priority list defined somewhere?
  - What's missing before the language can be considered ready for extended usage?
  - What's the vision for the language? Is it just a fun/hobby project, or is the ambition to see "general adoption" for it some day?

[redacted] 05/22/2024 10:24 AM

Hiya! I'm Nate - a Sys Admin in the UK - The idea of Amber is great! (edited)

[redacted] 05/22/2024 1:21 PM

Hi there. ^_^. Runtime Safety and Type Safety attracted my attention for the language and looking like interesting to track and try.
it is only gradual typing looking like, but still interesting approach.

1

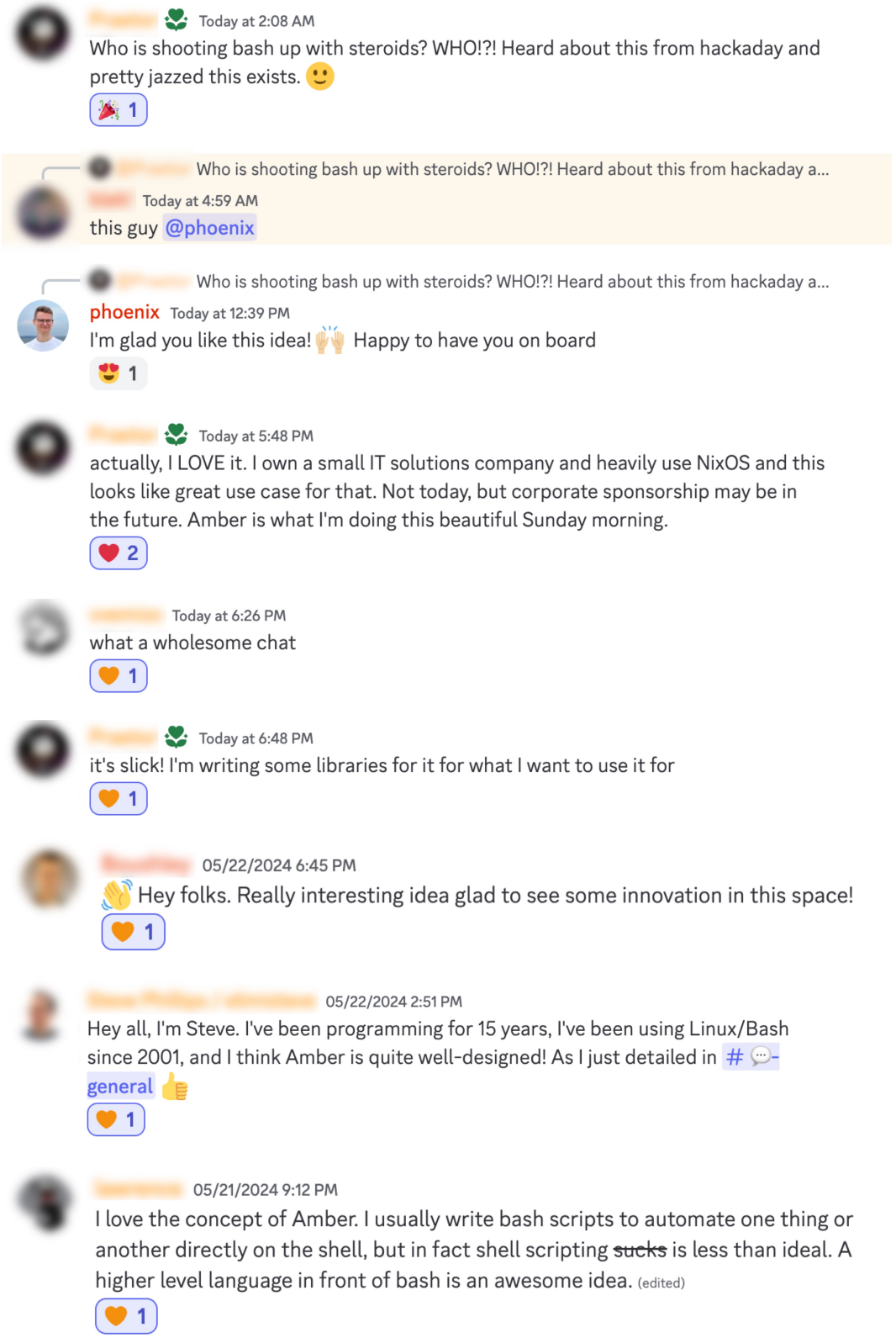
Amber tool reminded me about another tool that tries to handle similar issues for Yaml in infrastructure world. https://cuelang.org/
Cuelang addresses similar issues with Runtime Safety and Type Safety for yaml and has powerful features to import golang native k8s types and reuse them during manifests generations, also providing such nice features as code reusage via dependency reinstall to other repos
https://github.com/uhthomas/automata/blob/main/k8s/amour/cilium/cluster_role_list.cue some repo that uses it a lot.

1

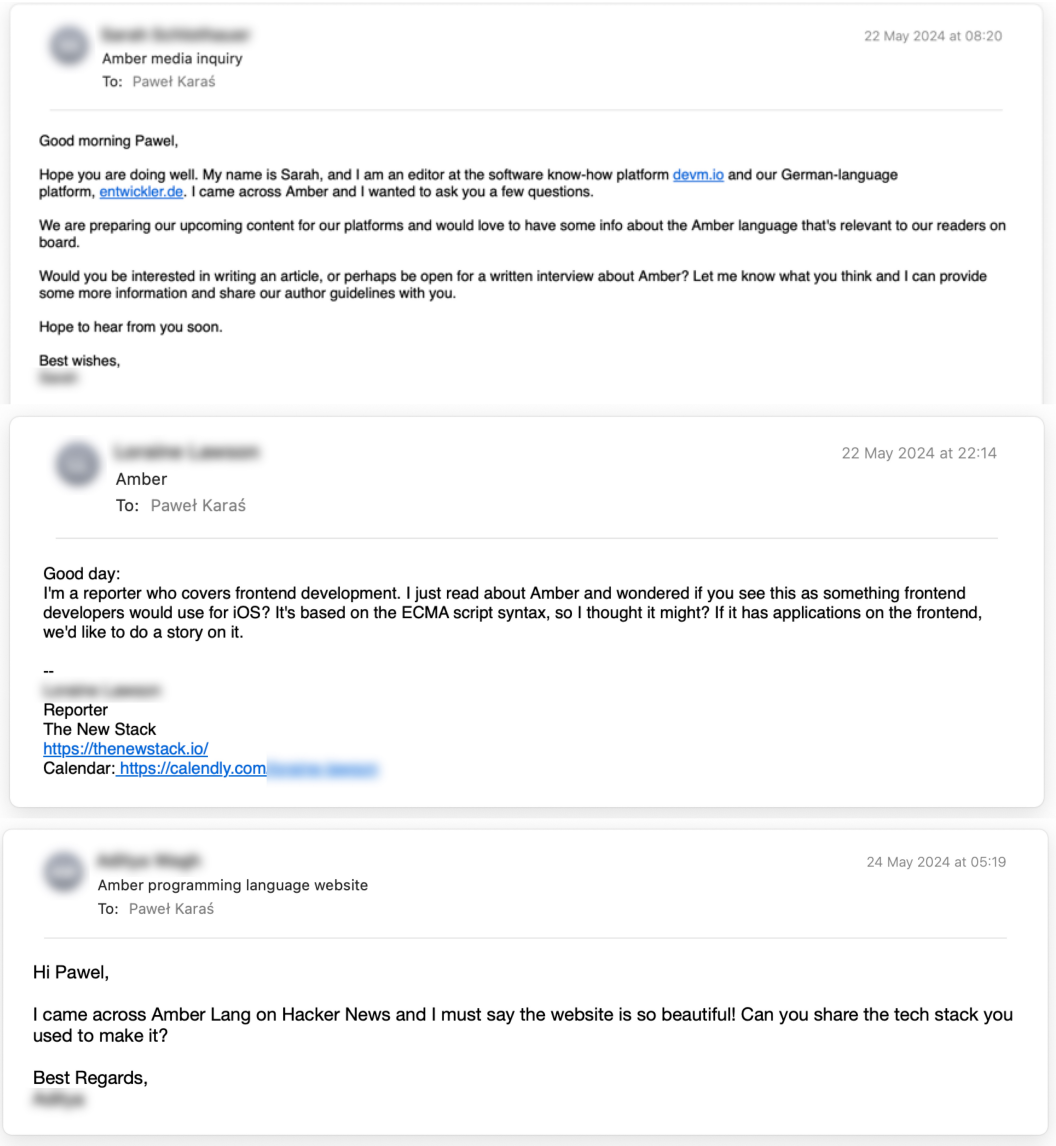
[redacted] 05/21/2024 8:58 PM

btw I am a physics student, finishing my bachelor this semester. I do not know rust, but do know C++ and python, so I can learn it. Really liked the idea of the project. I mostly use the fish shell myself, precisely for the simpler scripting and standard library. Amber seems to solve much of this!

1


Today at 2:08 AM
Who is shooting bash up with steroids? WHO!?! Heard about this from hackaday and pretty jazzed this exists.
1
Who is shooting bash up with steroids? WHO!?! Heard about this from hackaday a...
Today at 4:59 AM
this guy @phoenix
Who is shooting bash up with steroids? WHO!?! Heard about this from hackaday a...
phoenix Today at 12:39 PM
I'm glad you like this idea! Happy to have you on board
1
Today at 5:48 PM
actually, I LOVE it. I own a small IT solutions company and heavily use NixOS and this looks like great use case for that. Not today, but corporate sponsorship may be in the future. Amber is what I'm doing this beautiful Sunday morning.
2
Today at 6:26 PM
what a wholesome chat
1
Today at 6:48 PM
it's slick! I'm writing some libraries for it for what I want to use it for
1
05/22/2024 6:45 PM
Hey folks. Really interesting idea glad to see some innovation in this space!
1
05/22/2024 2:51 PM
Hey all, I'm Steve. I've been programming for 15 years, I've been using Linux/Bash since 2001, and I think Amber is quite well-designed! As I just detailed in # -general
1
05/21/2024 9:12 PM
I love the concept of Amber. I usually write bash scripts to automate one thing or another directly on the shell, but in fact shell scripting sucks is less than ideal. A higher level language in front of bash is an awesome idea. (edited)
1

7.5. Wiadomości email



[redacted]
Amber media inquiry
To: Paweł Karaś

22 May 2024 at 08:20

Good morning Pawel,

Hope you are doing well. My name is Sarah, and I am an editor at the software know-how platform devm.io and our German-language platform, entwickler.de. I came across Amber and I wanted to ask you a few questions.

We are preparing our upcoming content for our platforms and would love to have some info about the Amber language that's relevant to our readers on board.

Would you be interested in writing an article, or perhaps be open for a written interview about Amber? Let me know what you think and I can provide some more information and share our author guidelines with you.

Hope to hear from you soon.

Best wishes,
[redacted]

[redacted]
Amber
To: Paweł Karaś

22 May 2024 at 22:14

Good day:
I'm a reporter who covers frontend development. I just read about Amber and wondered if you see this as something frontend developers would use for iOS? It's based on the ECMA script syntax, so I thought it might? If it has applications on the frontend, we'd like to do a story on it.

--
[redacted]
Reporter
The New Stack
https://thenewstack.io/
Calendar: https://calendly.com[redacted]

[redacted]
Amber programming language website
To: Paweł Karaś

24 May 2024 at 05:19

Hi Pawel,

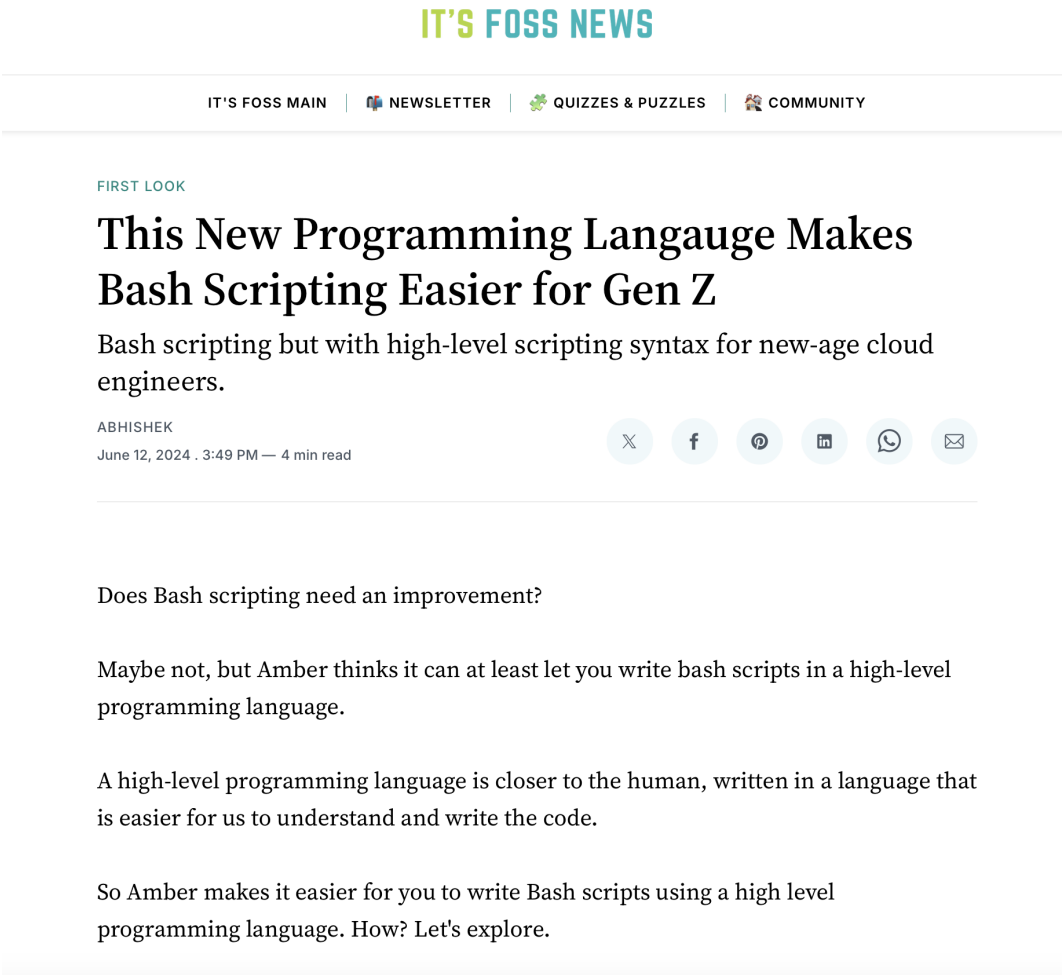
I came across Amber Lang on Hacker News and I must say the website is so beautiful! Can you share the tech stack you used to make it?

Best Regards,
[redacted]

7.6. Artykuły o języku Amber

Pojawiły się również artykuły na:

- It's FOSS - blog skupiony wokół darmowego i otwarto źródłowego oprogramowania.
- Hackaday - strona internetowa z artykułami o projektach DIY (Do It Yourself), elektronice i hackowaniu sprzętu.
- Zenn - japońska platforma do dzielenia się wiedzą techniczną, zawierająca artykuły i tutoriale.
- Dev.to - społeczność programistów, gdzie można publikować artykuły, dzielić się wiedzą i dyskutować na tematy związane z programowaniem.



IT'S FOSS NEWS

IT'S FOSS MAIN | NEWSLETTER | QUIZZES & PUZZLES | COMMUNITY

FIRST LOOK

This New Programming Langauge Makes Bash Scripting Easier for Gen Z

Bash scripting but with high-level scripting syntax for new-age cloud engineers.

ABHISHEK

June 12, 2024 . 3:49 PM — 4 min read

Does Bash scripting need an improvement?

Maybe not, but Amber thinks it can at least let you write bash scripts in a high-level programming language.

A high-level programming language is closer to the human, written in a language that is easier for us to understand and write the code.

So Amber makes it easier for you to write Bash scripts using a high level programming language. How? Let's explore.

Zenn

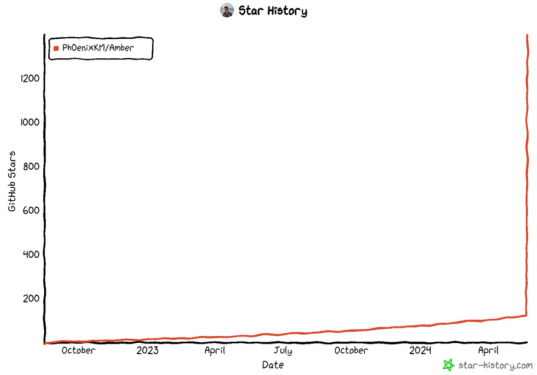
Bash shell amber Tech

シェルスクリプトを型安全に書きたいと考える人もいると思います。
そういった人におすすめのRust製プログラミング言語Amberを使ってみたので簡単に紹介します。

Amber The Programming Language
Amber The Programming Language
amber-lang.com

> Write your scripts in a modern type-safe and runtime-safe programming language that handles many bugs and mistakes during compilation process.

GitHub Star Historyによると最近一気にGitHubのスター数を獲得したようです。



arapower
フォロー

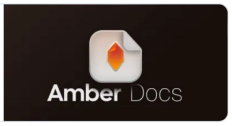
I think of myself as a DevOps person - nothing makes me happier than a battle-hardened shell script.

バッジを贈る
バッジを贈るとは →

目次

きむそん 6日前

Amber Documentation
Documentation for Amber programming language
docs.amber-lang.com



これを触っていく。現在Alpha版

返信

HACKADAY

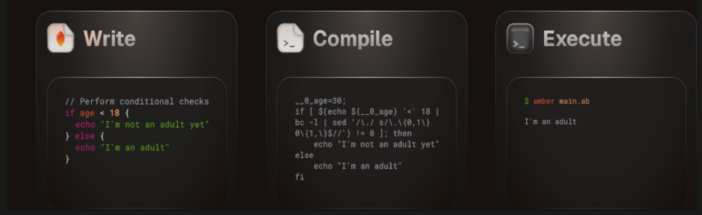
HOME BLOG HACKADAY.IO TINDIE HACKADAY PRIZE SUBMIT ABOUT June 14, 2024

AMBER COMPILES TO BASH

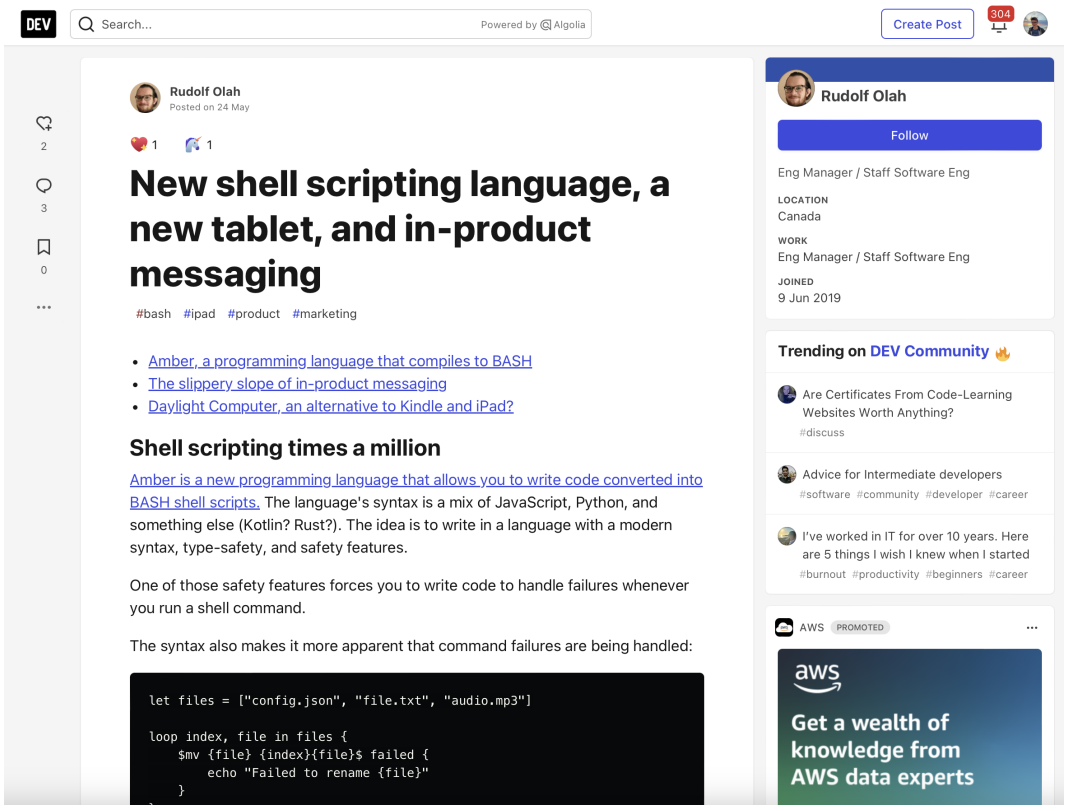
by: Al Williams

17 Comments

May 22, 2024



It certainly isn't a new idea to compile a language into an intermediate language. The original C++ compiler outputs C code, for example. Enhanced versions of Fortran were often just conversions of new syntax to old syntax. Of course, it makes sense to output to some language that can run on lots of different platforms. So, using that logic, Amber makes perfect sense. It targets — no kidding — bash. You write with nice modern syntax and compile-time checks. The output is a bash script. Admittedly, sometimes a hard-to-read bash script, but still.


DEV
Search...
Powered by Algolia
Create Post
304
Rudolf Olah
Posted on 24 May
2
3
0
1
1
New shell scripting language, a new tablet, and in-product messaging
#bash #ipad #product #marketing
Amber, a programming language that compiles to BASH
The slippery slope of in-product messaging
Daylight Computer, an alternative to Kindle and iPad?
Shell scripting times a million
Amber is a new programming language that allows you to write code converted into BASH shell scripts. The language's syntax is a mix of JavaScript, Python, and something else (Kotlin? Rust?). The idea is to write in a language with a modern syntax, type-safety, and safety features.
One of those safety features forces you to write code to handle failures whenever you run a shell command.
The syntax also makes it more apparent that command failures are being handled:
let files = ["config.json", "file.txt", "audio.mp3"]
loop index, file in files {
$mv {file} {index}{file}$ failed {
echo "Failed to rename {file}"
}
Rudolf Olah
Follow
Eng Manager / Staff Software Eng
LOCATION
Canada
WORK
Eng Manager / Staff Software Eng
JOINED
9 Jun 2019
Trending on DEV Community
Are Certificates From Code-Learning Websites Worth Anything?
#discuss
Advice for Intermediate developers
#software #community #developer #career
I've worked in IT for over 10 years. Here are 5 things I wish I knew when I started
#burnout #productivity #beginners #career
AWS
PROMOTED
aws
Get a wealth of knowledge from AWS data experts

7.7. Skrypty napisane przez innych użytkowników

7.7.1. Quick sort - autor UrbanCoffee

Jako, że Amber aktualnie nie wspiera wielowymiarowych tablic, widzimy, że programista wykorzystał ciągi znaków do przechowywania wielu liczb w tablicy. Implementacja funkcji wykonującej szybkie sortowanie [18]:

```
fun qs(ref qs_r_arr: [Num]): Null {
    let qs_arr = qs_r_arr
    let stack = [Text] // index range - "start end"
    stack += ["0 {len(qs_arr)-1}"]

    loop {
        if(empty(stack)): break
        let top = stack[-1]
        stack = pop(stack)

        let ranges = parse_split(top, " ") failed {
            fail 1 // SHOULDN'T happen
        }
        let left = ranges[0]
        let right = ranges[1]
        if (left >= right): continue

        // pivot index
        let pivot = floor(random() * (right - left)) + left
        swap(qs_arr, pivot, right)
        pivot = qs_arr[right] // pivot value

        let l = left
        let r = right - 1
        loop {
            if {
                l > r: break
                qs_arr[l] > pivot and qs_arr[r] < pivot {
                    swap(qs_arr, l, r)
                    l += 1
                    r -= 1
                }
                else {
```

```
                    l += (qs_arr[l] <= pivot then 1 else 0)
                    r -= (qs_arr[r] >= pivot then 1 else 0)
                }
            }
        }
        swap(qs_arr, l, right)
        stack += ["{left} {l-1}", "{l+1} {right}"]
    }
    qs_r_arr = qs_arr
}
```

7.7.2. Instalator serwerów LSP - autor Mte90

Poniżej znajduje się fragment instalatora, który pobiera najnowsze serwery LSP (Language Server Protocol) dla różnych języków programowania [16]:

```
fun get_download_path(repo, position) {
    return unsafe $curl -sL \
        https://api.github.com/repos/{repo}/releases \
        | jq -r ".[0].assets.[{position}].browser_download_url"$
}

silent unsafe $cd /tmp$

echo "Install PHPactor LSP"
let download_url = get_download_path("phpactor/phpactor", 0)
$wget {download_url}$ failed {
    echo "Error! Exit code: {status}"
}
silent unsafe {
    $mv phpactor.phar /usr/local/bin$
    $chmod +x /usr/local/bin/phpactor$
}

echo "Install Typos LSP"
let download_url = get_download_path("tekumara/typos-lsp", 6)
$wget {download_url}$ failed {
    echo "Error! Exit code: {status}"
}
```

```
silent unsafe {
    $tar -zxvf ./typos* -C ./typos-lsp$
    $mv typos-lsp /usr/local/bin$
    $chmod +x /usr/local/bin/typos-lsp$
}
// ...
```

7.7.3. Gra CLI Kółko i krzyżyk - autor MacioszekTV

Poniżej znajduje się fragment gry przedstawiający pętlę samej rozgrywki [13]:

```
pub fun game() {
    echo "\nPlayer 1 move:\n"
    echo "Select a row (1, 2, 3): "
    let selectedRow = parse(input()) failed {
        echo "\nInvalid input!"
        exit(1)
    }
    echo "Select a column (1, 2, 3): "
    let selectedCol = parse(input()) failed {
        echo "\nInvalid input!"
        exit(1)
    }
    let tableMove = move(selectedRow, selectedCol)
    table[tableMove] = "O"
    printTable()
    if isPlaying == true {
        echo "\nPlayer 2 move:\n"
        echo "Select a row (1, 2, 3): "
        let selectedRow2 = parse(input()) failed {
            echo "\nInvalid input!"
            exit(1)
        }
        echo "Select a column (1, 2, 3): "
        let selectedCol2 = parse(input()) failed {
            echo "\nInvalid input!"
            exit(1)
        }
        let tableMove = move(selectedRow2, selectedCol2)
```

```
            table[tableMove] = "X"
            printTable()
        }
    }

    loop {
        if isPlaying == false {
            echo "\n\nGame over!"
            echo "Winner is {winner}"
            break
        } else {
            game()
        }
    }
```

Rozdział 8.

Dalszy rozwój języka

W dalszym rozwoju języka Amber planuję skupić się na stabilizacji składni, optymalizacji kodu wyjściowego w języku Bash oraz rozwoju biblioteki standardowej. Pracuję również nad dodaniem następujących funkcji:

- `upper` - konwertuje ciąg znaków do wielkich liter.
- `lower` - konwertuje ciąg znaków do małych liter.
- `trim` - usuwa białe znaki z lewej i prawej strony ciągu znaków.
- `split` - dzieli tekst na osobne części, zwracając tablicę tekstu.
- `join` - łączy elementy tablicy tekstu w jeden ciąg znaków.
- `len` - zwraca liczbę elementów w tablicy.
- `sum` - liczy sumę elementów w tablicy liczb.
- `pop` - usuwa ostatni element z tablicy.
- `reverse` - odwraca kolejność elementów w tablicy.
- `input` - czyta dane dla standardowego wejścia.
- `replace` - zamienia wystąpienia danego podciągu w tekście na inny.
- `file_exists` - sprawdza czy plik istnieje.
- `dir_exists` - sprawdza czy katalog istnieje.
- `file_read` - otwiera plik i zwraca jego zawartość w kodowaniu UTF-8.
- `file_write` - zapisuje treść do pliku.
- `file_append` - dopisuje treść do pliku.
- `lines` - dzieli tekst na linie zwracając tablicę tekstu.

- `chars` - konwertuje tekst na tablicę znaków.
- `includes` - zwraca flagę czy tablica zawiera element.

Wśród planowanych funkcji znajduje się również wiele innych, które będą przydatne dla użytkowników. Wraz z rozwojem kompilatora rozpocznę pracę nad protokołem LSP (Language Server Protocol) dla tego języka, który ułatwi użytkownikom pisanie kodu w środowiskach programistycznych dodając takie funkcje jak: dokańczanie składni, podświetlanie błędów, wyświetlanie dokumentacji dla podświetlonych fragmentów kodu, przeglądanie symboli języka Amber oraz sugerowanie importów.

Planuję również dodać obsługę tablic wielowymiarowych. Przykładowo, aby zaimplementować tablicę dwuwymiarową, mogę wykorzystać następującą implementację w języku Bash:

```
arr0=(1 2 3)
arr1=(4 5 6)

# Nazwy zmiennych zawierające tablice
arr=(arr0 arr1)

# echo arr[0][1]
eval echo \${${arr[0]}[1]}
```

W dalszym etapie rozwoju planuję wdrożyć obsługę formatu JSON bez konieczności instalowania dodatkowych zależności, co umożliwi użytkownikom na jeszcze prostszą komunikację z serwerem przy pomocy tego formatu.

Rozdział 9.

Podsumowanie

Dzięki intensywnej (ponad dwuletniej) pracy nad projektem, udało się zrealizować wszystkie zamierzone cele oraz wprowadzić język Amber na rynek. Został on entuzjastycznie przyjęty przez społeczność programistyczną, zdobywając liczne pozytywne recenzje i wysokie oceny użytkowników na różnych platformach.

9.1. Osiągnięcia

Osiągnięcia projektu w czasie trwania kampanii (w okresie 23 dni).

Liczba Gwiazdek na platformie GitHub	3 058
Łączna liczba ściągnięć instalatora	2 266
Liczba wejść na stronę dokumentacji	41 352
Liczba wejść na stronę repozytorium	8 410
Całkowita liczba głosów na platformie Reddit	845
Całkowita liczba wyświetleń postów na platformie X	20 000+
Liczba punktów zdobytych na Hacker News	463
Liczba członków społeczności Amber na serwerze Discord	133

9.2. Przyszłość i rozwój

Plany rozwoju języka obejmują dalszą optymalizację kompilatora, rozszerzenie funkcjonalności oraz aktywne wsparcie społeczności użytkowników. Amber ma na celu ułatwienie pisania skryptów Bash, jednocześnie zwiększając ich bezpieczeństwo i czytelność, co czyni go wartościowym narzędziem dla szerokiego grona użytkowników. Osobiście jestem bardzo zadowolony z osiągniętych rezultatów i wierzę, że przyszłość języka Amber wygląda obiecująco.

Bibliografia

[1] J. Blandy, J. Orendorff, and L.F.S. Tindall. *Programming Rust*. O'Reilly Media, 2021.

[2] Richard Blum and Christine Bresnahan. *Linux command line and shell scripting bible*. Wiley,, New Delhi :, 3rd ed. edition, ©2015. Including index.

[3] Canonical. Landscape documentation: Explanation of ppa, 2024.
`https://ubuntu.com/landscape/docs/explanation-PPA`
Ostatni dostęp: 6 czerwca 2024.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[5] GNU. Bash source code: array.h, 2024.
`http://git.savannah.gnu.org/cgit/bash.git/tree/array.h`
Ostatni dostęp: 6 czerwca 2024.

[6] GNU. Bash source code: variables.h, 2024.
`http://git.savannah.gnu.org/cgit/bash.git/tree/variables.h`
Ostatni dostęp: 6 czerwca 2024.

[7] Paweł Karaś. Amber documentation, 2024.
`https://docs.amber-lang.com`
Ostatni dostęp: 6 czerwca 2024.

[8] Paweł Karaś. Amber syntax support extension, 2024.
`https://marketplace.visualstudio.com/items?itemName=PhOenixKM.amber-language`
Ostatni dostęp: 6 czerwca 2024.

[9] Paweł Karaś. Amber the programming language, 2024.
`https://amber-lang.com`
Ostatni dostęp: 6 czerwca 2024.

[10] Paweł Karaś. Amber: The programming language compiled to bash, 2024.
`https://github.com/PhOenixKM/Amber`
Ostatni dostęp: 6 czerwca 2024.

[11] Paweł Karaś. Heraclitus: The compiler frontend for developing great programming languages, 2024.
`https://github.com/Ph0enixKM/Heraclitus`
Ostatni dostęp: 6 czerwca 2024.

[12] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1997.

[13] MacioszekTV. Tic tac toe written in amber, 2024.
`https://github.com/MacioSzekTV/TicTacToe-Amber/blob/main/main.ab`
Ostatni dostęp: 14 czerwca 2024.

[14] MacroMates. Textmate manual: Language grammars, 2024.
`https://macromates.com/manual/en/language_grammars`
Ostatni dostęp: 6 czerwca 2024.

[15] Microsoft. Github: Where the world builds software, 2024.
`https://github.com`
Ostatni dostęp: 6 czerwca 2024.

[16] Mte90. Lsp installer written in amber, 2024.
`https://github.com/Mte90/My-Scripts/blob/master/dev/lsp-installer/install.ab`
Ostatni dostęp: 14 czerwca 2024.

[17] Mike McQuaid Rémi Prévost and Danielle Lalonde. Homebrew the missing package manager for macos (or linux), 2024.
`https://brew.sh`
Ostatni dostęp: 6 czerwca 2024.

[18] UrbanCoffee. Quick sort written in amber, 2024.
`https://github.com/UrbanCoffee/amber-projects/blob/main/collection/sorts/qs.ab`
Ostatni dostęp: 14 czerwca 2024.