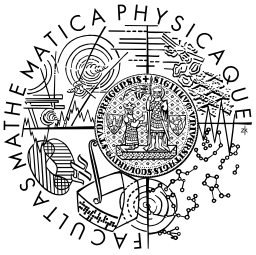


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Maximilian Kulikov

Emulátor zvukových syntezátorů

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Klusáček, Ph.D.
Studijní program: Informatika
Studijní obor: IOI

Praha 2022

Děkuji Mgr. Davidovi Klusáčkovi, Ph.D. za veškerou pomoc s mnohými teoretickými i praktickými otázkami v průběhu práce a nasměrování k výsledné podobě.

Název práce: Emulátor zvukových syntezátorů

Autor: Maximilian Kulikov

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Klusáček, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Nástroj na tvoření emulátorů zvukových syntezátorů. Základem práce je imperativní programovací jazyk Cynth popisující signály tvořící výsledný zvuk. Kód v jazyce Cynth se přeloží do jazyka C pro následující slinkování s programem řídícím GUI a MIDI vstupní ovládání a výstupní monitorování a napojení zvukové karty. Mezikrok s překladem do jazyka C přináší výhodu z využití optimalizací překladače C. Jazyk Cynth je omezený tak, aby za běhu nedocházelo k žádné dynamicé alokaci, ale zároveň umožňuje komplexní programování za překladu a práci se staticky alokovanými datovými strukrurami určenými k expresivnímu popisu signálů.

Klíčová slova: syntezátor emulátor programovací jazyk

Title: Sound Synthesizer Emulator

Author: Maximilian Kulikov

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. David Klusáček, Ph.D., Institute of Formal and Applied Linguistics

Abstract: A tool for creation of emulators of audio synthesizers. The base of the work is an imperative programming language Cynth that describes signals of the resulting sound. Cynth code is translated into C code for further linkage with a program that controls GUI and MIDI input controls and output monitoring and connection with a sound card. The intermediate step of translation into C allows taking advantage of the C compiler optimizations. The Cynth language is restricted in a way that eliminates any dynamic allocations at run-time while allowing complex compile-time programming and working with statically allocated data structures for expressive description of signals.

Keywords: synthesizer emulator programming language

Obsah

Úvod

Původním cílem bylo vytvořit nástroj nástroj na tvoření emulátorů zvukových syntezátorů v následujícím rozsahu: Programovací jazyk Cynth, který umožní popsat signály a deklarovat vstupy, výstupy a ovládací prvky. Kód v jazyce Cynth se přeloží do jazyka C pro následující statické slinkování s předkompilovaným řídícím programem. Tento řídící program měl mít na starosti komunikaci se zvukovou kartou a vykreslení GUI a napojení MIDI k ovládání a monitorování. V průběhu práce se ukázalo, že jde o mnohem komplexnější cíl, než jsem si původně představoval. Kompletní práci dle původního zadání se mi bohužel dokončit včas nepodařilo. Tato práce je omezena jen na uvedený konfigurační programovací jazyk a jeho překladač do jazyka C. Práce obsahuje popis implementace, neformální specifikaci jazyka, uživatelský manuál, uvedení vlastností a konstruktů jazyka Cynth, které byly pro uvedené napojení s řídícím programem navrženy a příklady implementace různých praktických signálů.

1. Seznámení s jazykem Cynth

Tato kapitola uvede základy jazyka Cynth ve stručné podobě. Tento popis jazyka je určen pro čtenáře, který má alospoň základní znalost jazyka C, nebo podobných či odvozených jazyků. Nejde o úplnou specikikaci jazyka. Ta je k nalezení v další kapitole.

1.1 Hello World!

Cynth aktuálně nemá možnost práce se stringy a ani k tomu není primárně určený. Namísto toho předvedu, jak vytvořit pole s ASCII hodnotami odpovídajícími textu „Hello World!“ a dále jak tyto hodnoty uvést do běhové části programu.

```
# ASCII characters of "Hello World!":
Int const [12] const helloWorld =
    Int const [12] ([
        72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33
    ]);
```

Komě komentáře `#...`, tento program obsahuje jediný příkaz, a to příkaz definice. Definice obsahuje typ, jméno a po rovnítku hodnotu. Uvádí novou proměnnou s daným jménem a typem a nastaví jí danou inicializační hodnotu.

Ve výše uvedeném programu jde konkrétně o definici proměnné `helloWorld` typu `Int const [12] const`, což označuje konstantní pole 12 konstantních hodnot celočíselného typu. Pole je vždy referenční, tedy odkazuje na nějaký soubor hodnot. Je-li pole konstantní, nelze ho „přesměrovat“ na jiný soubor hodnot. Jsou-li hodnoty pole konstantní, nelze tyto hodnoty měnit. *Je rozumné pokud možno volit konstantní typy, jelikož je překladač pak schopen provést více optimalizací. Takovou hodnotu překladač dokonce spočte za překladu a žádnou proměnnou ani žádný soubor hodnot za běhu nevytvoří.*

Napravo od rovnítka je literál pole. Literál pole vytvoří daný soubor hodnot a k němu i pole, tedy odkaz na ně. *Tyto hodnoty se alokují v rámci funkce, nebo v rámci celého programu. Veškeré operace s poli v Cynthu jsou omezeny tak, že nikdy nenastane zásah do již neexistujících hodnot.* Toto nově vytvořené pole je vždy nekonstantního typu, tedy `Int [12]`. V Cynthu neprobíhají implicitní konverze, až na přidání či odebrání konstantnosti, pokud nejde o konstantnost odkazovaných hodnot. V tomto případě je pro přiřazení potřeba přidat konstantnost odkazované hodnoty i samotného odkazu. Konstantnost odkazu není třeba řešit, ale konstantnost odkazovaných hodnot je třeba přidat ručně pomocí konverze. Konverze je dána výsledným typem a hodnotou ke konverzi v závorce.

Prorgam napsaný v Cynthu má za účel vytvořit nějaké struktury, o kterých popíše, jak se mají za běhu chovat, aby emulovaly zvukový syntezátor. Základní takovou strukturou je buffer. Buffer představuje pole, do kterého se za běhu zapisují hodnoty, a to tak, že se pro přidání nové hodnoty ta nejstarší hodnota zapomene. Zápis do bufferu nelze provádět ručně, jako u pole. Pro zápis je třeba

definovat speciální funkci, tzv. generátor, který se bude za běhu automaticky vyhodnocovat a výsledná hodnota zapisovat.

```
buffer [12] output = buffer[12] (Float fn (Int t) {
    return Float (t);
});
```

Jde opět o definici s tím, že do nové proměnné `output` přiřazujeme funkci (danou anonymní funkcí, neboli literálem funkce). Opět je třeba ručně provést konverzi, a to na buffer. Konverze z funkce na buffer vytvoří nový buffer a danou funkci zaregistruje jako generátor tohoto bufferu. Buffer je také referenční, takže dál se do nové proměnné předává odkaz na již existující buffer a žádný další se nevytváří. *Buffery se alokují v rámci celého programu.*

Takto definovaný generátor bere celočíselný (`Int`) prametr `t` a vrací ho jako reálné číslo (`Float`). Každý generátor musí být typu `Float (Int)` (tedy funkce z Intu do Floatu) nebo `Float ()` (funkce z „ničeho“ do Floatu). Výsldek generátoru reprezentuje vzorek nějakého signálu a jako vstup generátor dostane (pokud si řekne, že ho chce) čas daný počtem uběhlých vzorků od začátku běhu emulátoru.

Pokud takový emulátor spustíme a po chvíli zastavíme, na výstupu se objeví:

```
Buffer ‘output‘ stopped at 9 with:
165.0
166.0
167.0
168.0
169.0
170.0
171.0
172.0
173.0
174.0
175.0
176.0
```

Pozn.: Jde o debugovací představení stavu emulátoru, ve kterém skončil. Prozatím je to jediný způsob, jak nějaký běhový výstup z programu dostat. Dalším krokem v implementaci bude zařídit zápis těchto hodnot do zvukové karty.

Můžeme tedy takovým způsobem vypsat „HelloWorld!“.

```
Int const [12] const helloWorld =
    Int const [12] ([
        72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33
    ]);

buffer [12] output = buffer[12] (Float fn (Int t) {
    return Float(helloWorld[t % 12]);
});
```

Výstupem může být například:

```
Buffer ‘output‘ stopped at 3 with:
100,
33,
72,
101,
108,
108,
111,
32,
87,
111,
114,
108,
```

Jak je vidět, do bufferu se neustále zapisuje další hodnota z `helloWorld` pole a implmentačně to vypadá tak, že si buffer pamatuje pozici „začátku“, která se neustále pohybuje a pak vrací na začátek.

1.2 Deklarace, definice a přiřazení

Proměnné lze **deklarovat** bez uvedení hodnoty nebo **definovat** s expliciní inicializační hodnotou. Proměnné deklarované (bez expliciní inicializační hodnoty) jsou implicitně inticializovány na nulovou hodnotu. Do existujících proměnných lze **přiřadit**, pokud nejsou konstantní.

```
Int v1;     # Deklarace.
Int v2 = 1; # Definice.
v2 = 2;     # Přiřazení
```

Navíc je možné definovat typové aliasy.

```
type Buffer = buffer [128];
type Signal = Float (Int);
```

Pozor na počáteční písmena! Jména typů vždy začínají velkým písmenem a jména hodnot malým. Oba typy jmen mohou dále pokračovat libovolnými písmeny, číslicemi nebo podtržítky.

1.3 Přehled dle typů

1.3.1 Jednoduché typy

Základem typového systému Cynthu jsou **jednoduché typy**. Jsou to typy Bool, Int a Float. Tyto typy jsou podobné odpovídajícím typům v C až na sémantiku některých operací s nimi. Hodnotu každého z těchto typů je možné vytvořit odpovídajícím literálem. Tyto literály jsou opět podobné těm z C. Jsou jen navíc rozšířené o vědeckou notaci v Intech a vynechání nuly před, nebo po desetinné tečce ve Floatech.

```
Int   v1 = 1;
Int   v2 = 1e2;
Float v3 = 1.;
Float v4 = .1;
Float v5 = 1.1;
Float v6 = 1.1e2;
Float v7 = 1.e2;
Float v8 = .1e2;
```

S jednoduchými typy lze prvádět podobné operace, jako v C. Tabulka 1.1 uvádí přehled různých operátorů seřazených od nejvyšší precednece k nejnižší. Tedy např. operace násobení váže operandy silněji než sčítání, ale mezi násobením a dělením není rozdíl v precedneci.

Název	**Výraz**	**Poznámka**
plus	`+a`	
mínus	`-a`	
negace	`!a`	
umocnění	`a ** b`	Jen pro Floaty.
násobení	`a * b`	
dělení	`a / b`	Celočíselné dělení zaokrouhluje dolů.
modulo	`a % b`	Pro Inty i Floaty. Odpovídá dělení se zaokrouhlováním dolů.
sčítání	`a + b`	
odečítání	`a - b`	
ostré nerovnosti	`a < b a > b`	
neostré nerovnosti	`a <= b a >= b`	
rovnost	`a == b`	
nerovnost	`a != b`	
konjunkce	`a && b`	Short-circuit.
disjunkce	`a || b`	Short-circuit.

Tabulka 1.1: Přehled operátorů

Konverze meze jednoduchými typy jsou také podobné těm v C až na to, že konverze z Floatu do Intu zaokrouhluje dolů.

K jednoduchým typům je možné přidat konstantnost. Např.:

```
Int const v = 1;
v = 2; # Chyba.
```

1.3.2 Pole

Pole jsou z části popsána v předchozí sekci. Tabulka 1.2 uvádí varianty konstantnosti pole, jak již bylo naznačeno.

`T [n]`	Nekonstantní pole
`T [n] const`	Konstantní pole
`T const [n]`	Pole konstantních hodnot
`T const [n] const`	Konstantní pole konstantních hodnot

Tabulka 1.2: Přehled konstantnosti polí

Přiřazení do pole přenastaví jeho referenci, nikoliv obsažené hodnoty. Pro přiřazení do obsažených hodnot lze použít subscript. Vynechání indexu v subscriptu označuje výběr všech prvků polí, a je tak možné přiřadit do všech prvků hromadně a přitom nezměnit referenci.

```
Int [2] const ca = [1, 2];
ca    = [2, 3]; # Chyba
ca[1] = 3;      # OK
ca[]  = [2, 3]; # OK

Int const [2] cva = [1, 2];
cva    = [2, 3]; # OK
cva[1] = 3;      # Chyba
cva[]  = [2, 3]; # Chyba
```

Deklarace pole (bez expliciní inicializační hodnoty) vytvoří pole zaplněné nulami daného typu.

Pole je možné konvertovat jen na pole stejného typu hodnot a menší nebo stejné velikosti. Přitom k typu hodnot lze přidat konstantnost, ale nelze ji odebrat. Konstantnost reference je možné přidat i odebrat. Při žádné konverzi se pole nekopíruje, jde stále o předávání reference.

Prvky literálu pole mohou být dány jednoduchými výrazy, ale také jedním ze speciálních konstruktů **rozmezí** či **rozpětí**. Rozmezí udává aritmetickou posloupnost. Rozpětí „rozbalí“ hodnoty daného pole.

```
Int [3] a;
a = [0 to 3];                # [0, 1, 2]
a = [3 to 0];                # [3, 2, 1]
a = [0 to 6 by 2];           # [0, 2, 4]
Int [3] b = [...a];          # [0, 2, 4]
Int [7] c = [0, ...a, 0 to 3] # [0, 0, 2, 4, 0, 1, 2]
```

1.3.3 Buffery

Buffery jsou z části popsány v předchozí sekci. Jak již bylo řečeno, zápis do bufferů probíhá pomocí generátorů. Čtení pak probíhá pomocí subscriptu, jako u

pole. Indexuje se ale po zpátku, tedy od nuly do záporných čísel. Nula označuje nejnovější vzorek signálu a menší hodnoty označují starší vzorky. Vynechání indexu v subscriptu zkopíruje všechny vzorky bufferu do nového pole. Zápis skrze subscript není možný.

Konvertovat lze buffer jen na buffer stejné nebo menší velikosti. *Konverze funkce na buffer je popsána v předchozí sekci.*

1.3.4 Funkce

Funkce jsou hodnoty. Lze s nimi inicializovat proměnné, nebo je vracet, ale v aktuální verzi je nelze předávat do funkcí jako argument. Syntakticky jsou podobné funkcím v C.

```
Int f (Int x) {
    return x + 1;
};
```

Pozor ale na středník po zavřené závorce! Složené závorky mají totiž v Cynthu obecnější význam, než v C. Bez středníku by taková definice funkce mohla znamenat něco jiného, pokud by kód pokračoval. Funkce lze také vytvořit anonynmně, jak je popsáno v předchozí kapitole.

```
Int (Int) f = Int fn (Int x) {
    return x + 1;
};
```

Anonymní funkce obsahuje keyword `fn` místo názvu funkce a je to výraz. Funkce lze definovat i uvnitř jiných funkcí. Funkce také umí „zachytit“ proměnné z vnějšího scopu. Takové zachycení probíhá kopií, až na speciální staticky alokovoané typy (buffery a IO typy).

```
Int (Int) add (Int x) {
    return Int fn (Int y) {
        return x + y;
    }
};
Int add1  = add(1);
Int three = add1(2);
```

1.3.5 IO typy

IO typy v tomto úvodu vynchám, jelikož nejsou prakticky využitelné v aktuální verzi. Jsou ale do podrobna popsány v další kapitole.

1.4 N-tice

Každý výraz je v Cynthu ve skutečnosti n-ticí výrazů. Obecně je n-tice uzávorkovaný seznam položek (v tomto případě výrazů) oddělených čárkou. 1-tice je speciální případ, kdy n-tice nemusí (ale může) být uzávorkovaná. Uzávorkování

výrazů je tedy jen expliciní uzávorkování 1-tice. Tyto n-tice jsou vždy ploché. Uzávorkování uvnitř nemění jejich strukturu.

```
1;          # 1-tice
(1);        # 1-tice
(1, 2);     # dvojice
(1, 2, 3)   # trojice
(1, (2, 3)) # trojice
```

Stejně tak i typy a další konstrukty jsou n-ticemi.

```
type T1 = Int;          # 1-tice typů
type T2 = (Int);        # 1-tice typů
type T3 = (Int, Float); # dvojice typů
Int v1;                 # 1-tice deklarací
(Int v2, Float v3);     # dvojice deklarací
(Int, Int) v4;          # 1-tice deklarací s dvojicí typů
(a, b[1], c[])          # trojice cílů
```

0-tice jsou v Cynthu také využívány. Například aplikace (volání) funkce bez parametrů se tváří být identickým tomu v C, ale ve skutečnosti jde o aplikaci na 0-tici, neboli prázdnou hodnotu.

```
f();                # prázdná hodnota v aplikaci funkce
return ();          # prázdná hodnota ve vrácení z procedury
type F1 = Int ();   # prázdný vstupní typ funkce
type F2 = () ();    # prázdný vstupní i výstupní typ funkce
type F2 = void ();  # prázdný vstupní i výstupní typ funkce
Int fn () {};       # prázdná deklarace parametrů funkce
() fn (Int x) {};   # prázdný výstupní typ funkce
void fn (Int x) {}; # prázdný výstupní typ funkce
```

1-tice sice nemusí být obecně uzávorkovány, ale v některých případech syntaxe vynucuje explicitně uvávorkovanou n-tici.

```
() f (Int x) {};
() f Int x {}; # Chyba
f(1);
f 1; # Chyba
```

Prvky v n-ticích jsou sice jen odděleny čárkami, ale lze psát i poslední „trailing“ čárku.

```
Int fooBar (
    Int x,
    Int y,
    Int z, # OK
) {
    ...
}
```

1.5 Blocky

Konstrukt blocku (bloku) je také trochu zobecněn. Block totiž může být výrazem i příkazem. Je-li použit jako výraz, pak se vyhodnotí a dá nějakou výslednou hodnotu. Je-li použit jako příkaz, pak je sémanticky podobný klasickému blocku z C. Příkaz vrácení ve skutečnosti nevrací z funkce, ale z výrazového blocku. Vrácená hodnota se propaguje „ven“ skrze různé složené příkazy, jako if, while, a také skrze příkazové blocky, dokud nenarazí na výrazový block, a z toho se vrátí. V příkladech výše bylo zatajeno, že funkce ke své definici ve skutečnosti nepotřebuje složené závorky. Šlo jen o block použitý jako tělo funkce. Tělo funkce je v Cynthu dáno právě výrazem, nikoliv příkazem, nebo „speciálním konstruktem ve složených závorkách“, jak je tomu v C.

```
Int succ (Int x) x + 1;
succ(1); # 2

Int (Int) add (Int x) Int fn (Int y) x + y;
Int (Int) add1 = add(1);
add1(2); # 3

Int (Int) (Int) mul = Int (Int) fn (Int x)
    Int fn (Int y) x * y;
Int (Int) mul2 = mul(2);
mul2(2); # 4
```

Příkazy v blocku jsou narozdíl od C jen odděleny středníky, ne ukončeny. Poslední „trailing“ středník je ale také povolen.

```
{ Int a; Int b; }; # OK
{ Int a; Int b };  # OK
```

Celý program je ve skutečnosti blockem. Jen má implicitní složené závorky.

```
Int a;
Int b  # OK
# EOF
```

I z programu lze tedy vrátit. V aktuální verzi se vrácená hodnota zahodí, ale i tak je vrácení stále užitečné pro ukončení inicializace emulátoru. *V dalších verzích bude hodnota vrácená z programu použita jako informativní hodnota o stavu a výsledku inicializace.*

```
buffer [3] left = f;
if (mono)
    return;
buffer [3] right = f;
```

1.6 Řídící struktury

V Cynthu jsou řídící struktury podobné strukturám v C: if..else, while a for. While je prakticky identické tomu z C. If..else má rozdíl v tom, že neumožňuje vynechat else větev. K tomu je navíc speciální řídící struktura when. For se také liší. Nejde o klasický `for (...; ...; ...)` loop, ale o trochu „modernější“ `for (... in ...)`. Proto se taky v Cynthu přesněji označuje jako for..in loop. For..in loop umožňuje iterovat přes dané pole, nebo přes několik polí zároveň na způsob „zipu“. Žádný ekvivalent příkazům `continue` nebo `break` v Cynthu neexistuje. Větve (resp. tělo smyčky) těchto struktur se udávají příkazy (tedy i blockem, je-li to třeba).

```
Int i = 0;
while (i <= 5)
    i = i + 1;

if (i == 5)
    i = 0   # Středník zde není třeba, ale může tu být.
else {      # Větev daná blockem.
    i = 5;
};          # Na konci středník být musí, pokud kód pokračuje

when (i == 0)
    i = 1;

for (Int j in [1 to 5]) {
    i = i + j;
};

for (Int a in [0 to 5], Int b in [5 to 0])
    i = i + a * b;
```

Řídící struktury if..else a for..in je také možné použít jako výrazy. If..else pak figuruje jako ternární operátor v C a for..in představuje „map“ funkcionalitu. V obou případech je třeba větve (resp. tělo) uvést výrazem.

```
Int x;
x = if (x == 0) 1 else x;

Int [3] arr = [1 to 4];
Int [3] double = for (Int e in arr) e * 2;
```

1.7 Příklady

1.7.1 Komponenty syntezátoru

Nadefinujme si pomocné typy pro expresivnější a stručnější zápis:

```
type Time     = Int;
type Sample   = Float;
```

```
type Sig      = Sample (Time);
type ConstSig = Sample ();
Int const buffSize = 4;
type Buff   = buffer [buffSize];
type Single = buffer [1];
```

Dále vytvoříme dva signály:

```
Sample fn (Time t) Sample (t);
Sample fn (Time t) Float (t % 10) / 10.;
```

V obvyklé reálné situaci mohou být funkce signálů náročné na výpočet. Přitom je často potřeba využívat hodnot signálů ve více časových okamžicích. Např. pro vyfiltrování specifických frekvencí signálu je třeba provést konvoluci s implulzní odezvou nějakého filtru. To znamená provést násobení mezi n hodnotami z minulosti signálu a n hodnotami z odezvy filtru pro n řádově často ve stovkách i výš. Právě pro to jsou v Cynthu buffery. Definované signály necháme zapisovat do bufferů, abychom měli k dispozici historii těchto signálů bez opakovaných výpočtů.

```
Buff   b = Buff   (Sample fn (Time t) Sample (t));
Single c = Single (Sample fn (Time t) Float (t % 10) / 10.);
```

Do bufferu b se budou zapisovat hodnoty stále zvětšující se o 1. Do bufferu c se budou zapisovat hodnoty od 0 do 9 vzestupně.

Následně můžeme definovat různé komponenty syntezátoru.

```
ConstSig foo (Buff a, Single b) {
    return Sample fn () (a[-1] + a[0]) * b[0];
};

Sig bar (Buff a, Single b) {
    return Sample fn (Time t)
        if (t % 2 == 0) 0. else (a[-1] + a[0]) * b[0];
};
```

Standardní situací je, že komponenty skládají buffery do signálů. Uživatel komponenty si pak vytvoří vlastní výsledný buffer dle potřeby. Kdyby skládaly signály, mohlo by docházet ke zbytečně opakujícím se výpočtům. Kdyby vracely buffery, mohlo by docházet k tomu, že se zbytečně vytvoří buffer, který se nakonec nevyužije, třeba protože u něj není potřeba zasahovat do minulosti.

```
Buff out left  = Buff (foo(b, c));
Buff out right = Buff (bar(b, c));
```

1.7.2 Konvoluce

Uvedu příklad jak spočítat typickou konvoluci dvou signálů daných buffery:

```
ConstSig conv (Buff a, Buff b) {
    Int result = 0;
    for (Int i in [0 to -buffSize], Int j in [-buffSize + 1, 1])
        result = result + a[i] * b[j];
    return result;
}
```

V reálné situaci by molo jít třeba i o konvoluci signálu si pevně danou implulzní odezvou filtru. Takovou implulzní odezvu by mohlo být vhodnější mít předpočítanou v poli.

```
ConstSig filter (Buff a, Int const [buffSize] const f) {
    Int result = 0;
    for (Int i in [0 to -buffSize], Int j in [-buffSize + 1, 1])
        result = result + a[i] * f[j];
    return result;
}
```

1.8 Spouštění a testování

Aktuálně relevantní testovací soubory jsou umístěny v adresáři `tests/misc/`. Jde o soubory s Cynth kódem s příponou `.cth`. K přeložení lze využít následující příkaz:

```
make tests/out/{name}
```

Místo `{name}` je třeba uvést zbytek cesty ke spustitelnému souboru k vytvoření. Např. `misc/001` přeloží kód ze souboru `tests/misc/001.cth` do spustitelného programu a umístí ho do `tests/out/001`.

V souboru `config.mk` lze nastavit paramerty pro překladač výsledného C kódu.

K překladu testovacích souborů jen do mezivýsledku v C, lze spustit např. následující příkaz:

```
cat tests/{name}.cth | dist/compiler
```

2. Specifikace

Tato kapitola obsahuje podrobnou specifikaci syntaxe a sémantiky jazyka Cynth. Syntaxe je popsána formální gramatikou, zatímco sémantika je popsána neformálně. Místy se objeví i komentář o relevantních implementačních detailech. Z většiny jsou konstrukty uváděny postupně, ale občas je třeba naznačit existenci nějakého konstruktu s tím, že se podrobněji definuje a popíše později.

2.1 Syntaxe

Syntaxe jazyka je při překladu analyzována ve dvou fázích, a to lexerem a parserem. Lexer vstupní text rozdělí na lexikální prvky, tzv **tokeny**, se kterými dále pracuje parser. Parser z přečtených tokenů sestaví AST (abstract syntax tree), které se dále transformují do pomocných sémantických struktur nebo do výsledného kódu v jazyce C.

2.1.1 Tokeny

Mezi tokeny patří několik tzv. **klíčových slov** (neboli **keywordů**). Ty jsou definovány jako řetězce písmen necitlivé na velikost písmen. Uvedu je výčtem:

```
buffer by const
else false fn
for if in
out return self
to true type
void when while
```

`self` je jen vyhrazeno pro další verze (k reprezentaci rekurzivního volání funkce.) Dále jde o tzv. **symboly** definovné jako řetězce speciálních znaků. Opět uvedu výčtem:

```
( ) [ ] { }
+ - * / % **
! && ||
== != >= <= > <
, ; = $ ...
```

Symbol **auto** (`$`) je vyhrazen pro odvození typu, ale není implementován v této verzi.

Zbylé tokeny jsou specifické řetězce definovné regulárními výrazy. Uvedu je výčtem názvů tokenů a odpovídajících regulárních výrazů:

```
blank     [ \t]
endl      [\n\r]
comment   ("//"|#).*$
multicom  "/*"([^*]|(\*+[^*/]))*\*+\/
name      [a-z][a-zA-Z0-9_]*
type_name [A-Z][a-zA-Z0-9_]*
```

```
int       [0-9]+([eE][+-]?[0-9]+)?
float     ([0-9]+\.[0-9]*|\.[0-9]+)([eE][+-]?[0-9]+)?
```

Merezy, konce řádků a komentáře (`blank`, `endl`, `comment`, `multicom`) parser ignoruje. Ostatní tokeny z výčtu výše nesou sémantickou hodnotu danou odpovídajícími řetězcemi, se kterou parser dál pracuje.

Jména (`name`), resp. jména typů (`type_name`), reprezentují názvy proměnných, resp. typů. Pro jednodušší analýzu jsou rozlišeny počátečním znakem. Jména začínají malým písmenem, zatímco jména typů začínají velkým písmenem. Jména, ani jména typů se nemohou shodovat s žádným z klíčových slov. `int`, `float` a `string` reprezentují literály odpovídajících typů. Číselné literály (`int` i `float`) umožnují zápis ve vědecké notaci. `float` umožňuje vynechat část před, nebo po desetinné tečce.

Jsou podporovány řádkové komentáře `//...` a `#...` a také víceřádkové `/*...*/` ve stylu C. Víceřádkové komentáře nepodporují vnořené komentáře stejného typu. *Pro podporu takového vnoření by byl potřeba zásobníkový automat při lexikální analýze a pouhé regulární výrazy by pro popis takového tokenu nestačily.*

2.1.2 Gramatika

Gramatiku uvedu ve formátu Bisonu s drobnými zjednodušeními v zápisu terminálů.

Gramatika obsahuje následující terminály: `int`, `float`, `string`, `name`, `type_name` a znaky uvozené v `‘...‘`. Terminály mohou být pro stručnost spojeny do jednoho uvození `‘...‘` a odděleny jen mezerou. Vše ostatní jsou neterminály.

Kořen:

```
start: empty | stmt_list | stmt_list ‘;‘
```

Sémantické kategorie: (Odpovídají sémantickým prvkům.)

```
cat_declaration: node_declaration | paren_decl
cat_range_decl: node_range_decl | paren_range_decl
cat_array_elem: node_range_to | node_range_to_by | node_spread |
    cat_expression
cat_type: node_auto | node_type_name | node_function_type |
    node_array_type | node_buffer_type | node_const_type |
    node_in_type | node_out_type | paren_type
cat_expression: expr_or | expr_right
cat_statement: pure | cat_expression
```

Syntaktické kategorie: (Neodpovídají žádným sémantickým prvkům, pouze zajišťují správné přednosti operátorů.)

```
pure: node_declaration | node_definition | node_assignment |
    node_type_def | node_function_def | node_return | node_if |
    node_for | node_while | node_when
expr_or: node_or | expr_and
expr_and: node_and | expr_eq
expr_eq: node_eq | node_ne | expr_ord
```

```
expr_ord: node_ge | node_le | node_gt | node_lt | expr_add
expr_add: node_add | node_sub | expr_mul
expr_mul: node_mul | node_div | node_mod | expr_pow
expr_pow: node_pow | expr_pre
expr_pre: node_minus | node_plus | node_not | expr_post
expr_post: node_application | node_conversion | node_subscript |
    expr_atom
expr_atom: node_name | node_bool | node_int | node_float | node_string |
    node_block | node_array | paren_expr
expr_right: node_expr_if | node_expr_for | node_function
expr_assgn_target: expr_post
```

Typy:

```
paren_type: ‘(‘ cat_type ‘)‘ | ‘(‘ type_list ‘)‘ | ‘(‘ type_list ‘, )‘
void_type: ‘( )‘ | ‘void‘
node_auto: ‘$‘
node_type_name: type_name
node_const_type: cat_type ‘const‘
node_in_type: cat_type ‘in‘
node_out_type: cat_type ‘out‘
node_function_type: cat_type paren_type | void_type paren_type |
    cat_type void_type | void_type void_type
node_array_type: cat_type ‘[‘ cat_expression ‘]‘ | cat_type ‘[ $ ]‘ |
    cat_type ‘[ ]‘ | cat_type ‘[‘ cat_declaration ‘]‘
node_buffer_type: ‘buffer[‘ cat_expression ‘]‘
```

Deklarace:

```
paren_range_decl: ‘(‘ cat_range_decl ‘)‘ | ‘(‘ range_decl_list ‘)‘ |
    ‘(‘ range_decl_list ‘, )‘
paren_decl: ‘(‘ cat_declaration ‘)‘ | ‘(‘ decl_list ‘)‘ |
    ‘(‘ decl_list ‘, )‘
void_decl: ‘( )‘
node_declaration: cat_type node_name
node_range_decl: cat_declaration ‘in‘ cat_expression
```

Speciální prvky polí:

```
node_range_to: cat_expression ‘to‘ cat_expression
node_range_to_by: cat_expression ‘to‘ cat_expression ‘by‘ cat_expression
node_spread: ‘...‘ cat_expression
```

Literály:

```
node_bool: ‘true‘ | ‘false‘
node_int: int
node_float: float
node_string: string
node_function: cat_type ‘fn‘ paren_decl cat_expression |
    void_type ‘fn‘ paren_decl cat_expression |
    cat_type ‘fn‘ void_decl cat_expression |
    void_type ‘fn‘ void_decl cat_expression
node_array: ‘[ ]‘ | ‘[‘ array_elem_list ‘]‘ | ‘[‘ array_elem_list ‘, ]‘
```

Operátory:

```
node_or: expr_or ‘||‘ expr_and
node_and: expr_and ‘&&‘ expr_eq
node_eq: expr_eq ‘==‘ expr_ord
node_ne: expr_eq ‘!=‘ expr_ord
node_ge: expr_ord ‘>=‘ expr_add
node_le: expr_ord ‘<=‘ expr_add
node_gt: expr_ord ‘>‘ expr_add
node_lt: expr_ord ‘<‘ expr_add
node_add: expr_add ‘+‘ expr_mul
node_sub: expr_add ‘-‘ expr_mul
node_mul: expr_mul ‘*‘ expr_pow
node_div: expr_mul ‘/‘ expr_pow
node_mod: expr_mul ‘\%‘ expr_pow
node_pow: expr_pre ‘**‘ expr_pow
node_minus: ‘-‘ expr_pre
node_plus:  ‘+‘ expr_pre
node_not:   ‘!‘ expr_pre
node_application: expr_post paren_expr | expr_post ‘void‘
node_conversion: cat_type paren_expr
node_subscript: expr_post ‘[‘ array_elem_list ‘]‘ | expr_post ‘[ ]‘
node_expr_if: ‘if‘ paren_expr cat_expression ‘else‘ cat_expression
node_expr_for: ‘for‘ paren_range_decl cat_expression
```

Ostatní výrazy:

```
paren_expr: ‘(‘ cat_expression ‘)‘ | ‘(‘ expr_list ‘)‘ |
    ‘(‘ expr_list ‘, )‘
void: ‘( )‘
node_name: name
node_block: ‘{ }‘ | ‘{‘ stmt_list ‘}‘ | ‘{‘ stmt_list ‘; }‘
```

Příkazy:

```
node_definition: cat_declaration ‘=‘ cat_expression
node_assignment: expr_assgn_target ‘=‘ cat_expression
node_type_def: ‘type‘ node_type_name ‘=‘ cat_type
node_function_def: cat_type node_name paren_decl cat_expression |
    void_type node_name paren_decl cat_expression |
    cat_type node_name void_decl cat_expression |
    void_type node_name void_decl cat_expression
node_return: ‘return‘ cat_expression | ‘return void‘ | ‘return‘
node_if: ‘if‘ paren_expr pure ‘else‘ pure |
    ‘if‘ paren_expr pure ‘; else‘ pure |
    ‘if‘ paren_expr cat_expression ‘else‘ pure |
    ‘if‘ paren_expr cat_expression SEMI ‘else‘ pure |
    ‘if‘ paren_expr pure ‘else‘ cat_expression |
    ‘if‘ paren_expr pure SEMI ‘else‘ cat_expression
node_when: ‘when‘ paren_expr cat_statement
node_for: ‘for‘ paren_range_decl pure
node_while: ‘while‘ paren_expr cat_statement
```

Pomocné struktury:

```
array_elem_list: cat_array_elem | array_elem_list ‘,‘ cat_array_elem
stmt_list: cat_statement | stmt_list ‘;‘ cat_statement
type_list: cat_type ‘,‘ cat_type | type_list ‘,‘ cat_type
expr_list: cat_expression ‘,‘ cat_expression |
    expr_list ‘,‘ cat_expression
decl_list: cat_declaration ‘,‘ cat_declaration |
    decl_list ‘,‘ cat_declaration
range_decl_list: cat_range_decl ‘,‘ cat_range_decl |
    range_decl_list ‘,‘ cat_range_decl
```

V zásadě je Cynth na první pohled syntakticky podobný jazyku C. Podoba s jazykem C ale nebyla zásadním kritériem při návrhu syntaxe. Spíše jsem se snažil o zobecnění syntaxe C do něčeho více konzistentního, bez zbytečných výjimek z obecných pravidel.

Celý program je **výraz**, konkrétně výraz block (`node_block`) s implicitními složenými závorkami. Block je výraz, který je tvořen složenými závorkami uzávorkovanými a středníky oddělenými **příkazy**. Příkazy zahrnují i výrazy. Každý výraz je **n-ticí** výrazů. N-tice jsou tvořeny uzávorkovaným seznamem výrazů oddělených čárkami, nebo, v případě 1-tice, samotným neuzávorkovaným výrazem. N-tice jsou ploché, tedy n-tice obsahující právě dvě dvojice je čtveřicí, stejně jako n-tice obsahující právě 4 1-tice. Uzávorkování v n-tici nemění její sémantiku. Velikost n-tice (tedy n) je nazývána její **aritou**. To samé platí pro n-tice **typů**, n-tice deklarací a n-tice **cílů**.

Speciálním případem jsou 0-tice. Výskyt 0-tic je syntakticky omezen na několik specifických případů, kde mají smysl. 0-tice se také nazývají prázdné hodnoty (výrazy), typy či deklarace. 0-tice cílů nejsou povoleny. Navíc je na zkoušku umožněna alternativní syntaxe prázdných výstupních typů funkcí s klíčovým slovem `void` pro případ, že by se syntaxe s prázdnými závorkami osvědčila jako příliš matoucí pro uživatele zvyklé na exvivlentní syntaxi v C.

V některých případech musí být n-tice explicitně uzávorkovány, i když jde o 1-tici. V gramatice tomu odpovídají neterminály s prefixem `paren_` (např. `paren_decl`).

2.2 Sémantika

Program je složený z deklarací, typů, příkazů, výrazů, cílů a prvků polí. Mezi příkazy kromě čistě příkazových konstruktů patří deklarace a výrazy. Příkazy se **vykonávají** (neboli **exekuují**). Výrazy se **vyhodnocují** (neboli **evaluují**). Vyhodnocený výraz má vždy tzv. **výslednou hodnotu**, Pojmy výraz a hodnota jsou záměnné. (Výraz je pojem syntaktický, hodnota sémantický.) Vykonaný příkaz může (ale nemusí) mít tzv. **návratovou hodnotu**. Zkráceně řečeno: výraz „**má hodnotu**“ a příkaz „**vrací**“. Výsledné i návratové hodnoty jsou dány n-ticemi. Vykonání příkazu tvořeným výrazem znamená vyhodnocení výrazu bez použití jeho hodnoty. Takto vykonané příkazy nemají návratovou hodnotu. Sémantiku vykonávání deklarací a čistě příkazových konstruktů popíšu dále individuálně. Sémantika cílů a prvků polí je specifická pro příkaz přiřazení a výraz literálu pole, které jsou popsány dále.

V následujícím textu budu uvádět různé pojmy s co nejpřesnějšími, ale nefor-

málními definicemi. Pokud např. nějaký pojem bude definován jako vztah mezi dvěma typy a využiju ho pro popis vztahu mezi dvěma hodnotami, je myšlen vztah dle původní definice, jen na typech odpovídajících uvedeným hodnotám. Pokud zmíním, že „příkaz vrací, pokud vrací vnořený příkaz“, je tím myšleno, že se vrací přesně stejnou hodnotu, stejného typu.

2.2.1 Uzávorkování

Operace **uzávorkování** je sémanticky identitou. Jde o explicitně uzávorkovanou 1-tici. Je syntakticky i sémanticky ekvivalentní pro hodnoty, typy, deklarace i cíle. Explicitní uzávorkování je někdy vynuceno syntaxí.

2.2.2 Typy

Každý výraz má určený **typ**, resp. n-tici typů.

Základem jsou tzv. **jednoduché typy**: **Bool**, **Int**, **Float**. Hodnoty typu Bool reprezentují binární booleovské hodnoty, které lze vytvořit literály `true` a `false`. Hodnoty typu Int reprezentují celá čísla. Jsou omezeny shora i zdola hodnotami danými implementací. Sémantika přesahu krajních hodnot je také dána implementací. Tyto hodnoty lze vytvořit **celočíselným literálem** (token `int`). Typ Float reprezentuje reálná čísla. Implementačně jde o floating-point hodnotu, tedy opět je třeba počítat s omezenými hodnotami a také s omezenou přesností. Float hodnoty lze vytvořit **destinným literálem** (token `float`).

V aktuální verzi je Int implementován pomocí typu int v C++ compile-time interpreteru i v C run-time kódu, velikost kterého závisí na platformě. V C++ je dvojkový doplněk zaručen. V C tomu tak být nemusí. Vzhledem k omezení cílové platformy na moderní verze Windows a překladače Clang a GCC se však lze spolehnout na dvojkový doplněk a velikost alespoň 32 bitů. Float je implementován typem float v obou prostředích. Standardní IEEE floaty nejsou zaručeny standardem, ale opět vzhledem k omezené cílové platformě se s tím dá počítat. V dalších verzích bude vhodné přidat konfiguraci požadavků na tyto typy v implementaci.

Hodnoty jednoduchých typů se **předávají** tzv. **hodnotou** (neboli **kopií**). Předání je libovolné použití hodnoty výrazu kromě tzv. **zachycení** ve funkci a **vrácení** z funkce. To znamená, že při předávání se jejich hodnota kopíruje, a ta původní zůstává nedotčená a z nové zkopírované hodnoty nepřístupná.

Dále jsou k dispozici tzv. **referenční typy**, které se předávají **referencí** (neboli **odkazem**). To znamená, že tyto hodnoty reprezentují **referenci** (**odkaz**) na jinou hodnotu. Samotná reference se předává hodnotou a nelze vytvořit další referenci odkazující na ní. Odkazovaná hodnota se ale při předávání této reference nekopíruje. Skrze odkaz lze hodnotu modifikovat.

Pole (neboli **array**) reprezentuje referenci na pevně daný počet jednoduchých hodnot umístěných v daném pořadí. Velikost pole je dána kompilační konstantou. V aktuální verzi jsou pole omezena na obsažení 1-tic jednoduchých typů. Typům polí v gramatice odpovídá neterminál `node_array_type`. Určuje ho typ odkazované hodnoty a počet odkazovaných (neboli **obsažených**) hodnot. **Vstupní**

(neboli **input**), resp. **výstupní** (neboli **output**), typy (zkráceně **IO typy**) reprezentují referenci na hodnotu, kterou lze jen číst, resp. jen modifikovat. Těmto typům odpovídají neterminály `node_in_type` a `node_out_type`. Určuje ho typ obsažené hodnoty. IO typy obahující n-tice jsou ekvivalentní n-ticím IO typů obahujících odpovídající 1-tice. **Buffery** reprezentují referenci na peveně daný počet hodnot typu Float (1-tice) umístěných v daném pořadí, které se za běhu cyklicky protáčí. Velikost bufferu je dána kompilační konstantou. Bufferům odpovídá neterminál `node_buffer_type`. Určuje ho jen počet obsažených hodnot.

Mimo roztřídění na jednoduché a referenční typy leží typ funkce. Funkce jsou hodnoty a z pohledu uživatele se předávají kopií. Nejsou ale považovány za jednoduché typy kvůli odlišným pravidlům mutability. Funkce představují zobrazení mezi hodnotami se side effecty, tedy obsahují spustitelnou paramertizovatelnou část programu, která může při spuštění ovlivnit zbytek programu.

IO typy a buffery jsou spojeny pod pojem **statické typy**. Tyto typy mají speciální sémantiku, pokud jde o jiné použití hodnoty než předávání.

K typům lze navíc specifikovat, zda jde o **imutabilní** (neboli **konstantní**) hodnotu přidáním keywordu **cost** za daný typ. Tomu odpovídá neterminál gramatiky `node_const_type`. Jednoduché typy mohou být konstantní i nekonstantní. Pole mohou mít konstantní i nekonstantní referenci na konstantní i nekonstantní hodnoty. Imutabilita reference se určí keywordem const za celým typem funkce, zatímco imutabilita hodnot se určí keywordem const za typem hodnot uvnitř celého typu pole. IO typy mají konstantní referenci na nekonstantní hodnoty. Přidání keywordu const za typ hodnoty je chybou, zatímco přidání keywordu const za celý IO typ je zbytečné, ale není chybou. Pro buffery platí to samé. Přidání keywordu const za typ bufferu je zbytečné, ale ne chybné. Funkce jsou imutabilní. Přidání keywordu const je opět zbytečné, ale není chybou.

U typů se uvažují dvě ekvivalence: identita a shoda. Shodné typy jsou identické až na mutabilitu. V případě jednoduchých typů je Bool shodný s Boolem, Int s Intem a Float s Floatem nehledě na mutabilitu. U bufferů, IO typů a funkcí pojmy identita a shoda splývají, jelikož mají pevně uerčenou mutabilitu. Pole jsou shodná, pokud jsou identické obsažené typy i počet obsažených hodnot, nehledě na mutabilitu reference. Tedy konstantní pole konstantních hodnot je shodné se stejně velkým nekonstantním polem konstantních hodnot, ale pole konstantních hodnot a pole nekonstantních hodnot se neshodují.

2.2.3 Definice

Odpovídající neterminál v gramatice: `node_definition`.

Definice je příkaz složený z deklarací reprezentujích definované proměnné a výrazů reprezentujích jejich **inicializační hodnoty**. Vykonání definice provede deklaraci se stejnou sémantikou jako příkaz deklarace a navíc deklarovaným proměnným nastaví uvedené hodnoty. Definovat lze proměnné všech typů. Inicializační hodnoty se musí shodovat s typem uvedeným v deklaraci. Je tedy možné definovat konstantní hodnotu z nekonstantní, ale není možné definovat pole konstantních

hodnot z pole nekonstantních hodnot.

Definice nikdy nevrací.

2.2.4 Block

Odpovídající neterminál v gramatice: `node_block`.

Block (neboli **blok**, obzvlášť při skloňování) je výraz, či příkaz, který obsahuje seznam tzv. **vnitřních příkazů**. Block je výrazem, pokud se nachází na místě, kde gramatika očekává výraz, nikoliv příkaz. V opačném případě je čistě příkazovým konstruktem. Vyhodnocení bloku v obou případech znamená vytvoření **scopu** a vykonání všech vnitřních příkazů v uvedeném pořadí. Scope bude podrobněji definován dále. Výrazový block se navíc vyhodnotí s výslednou hodnotou danou návratovou hodnotou prvního vykonaného vnitřního příkazu, který vrací. Pokud žádný takový příkaz ve výrazovém bloku není, jedná se o sémantickou chybu. Příkazový block se vykoná s návratovou hodnotou danou prvním vnitřním příkazem, který vrací. Příkazový block vrací vždy, pokud obsahuje vnitřní příkaz, který vrací vždy. Příkazový block nevrací, pokud neobsahuje žádný vnitřní příkaz, který vrací. Všechny vnitřní příkazy se musí shodovat v typu návratové hodnoty. Příkazy po prvním vždy vracejícím příkazu se ignorují. *V dalších verzích bude vhodné alespň kontrolovat správnost této ignorované části programu.*

Celý program je definován jako block se syntakticky implicitními složenými závorkami. Tento block se označuje jako **nejvnější** (**outermost**) neboli **globální** block. Označení globální je zavedeno kvůli podobnosti s C, ale ve skutečnosti není nijak sémantiky odlišný od ostatních bloků, proto preferuji pojem nejvnější.

2.2.5 Return

Odpovídající neterminál v gramatice: `node_return`.

Return, neboli **vrácení**, je příkaz, který vždy vrací uvedenou hodnotu. Return může vracet i prázdnou hodnotu. Pokud se hodnota neuvede (`return`), implicitně se vrátí prázdná hodnota (`return ()`).

Jelikož celý program je výrazový block, je možné vrátit i z programu. Vykonání příkazu return (mimo nějaký jiný výrazový block než ten nejvnější) vrátí výslednou hodnotu programu. Cílem programu vytvořeném v Cynthu ale není vrátit jedinou hodnotu. Cílem je nadefinovat buffery a IO typy pro emulaci syntezátorů. Tato návratová hodnota je ale užitečná jako výsledek inicializace syntezátoru. *V aktuální verzi není implimentován žádný feedback na vrácenou hodnotu. Vrácení z programu je ale stále užitečné alespoň pro zastavení inicializace syntezátoru.*

2.2.6 Deklarace

Odpovídající neterminál v gramatice: `node_declaration`.

Stejně jako typy a hodnoty jsou i deklarace n-ticemi. 1-tice deklarace je dána typem (resp. n-ticí typů) a jménem. Takové deklarace lze skládat do libovolných n-tic. Deklarace uvádí pojmenované proměnné obsahující hodnotu daného typu. Proměnné jsou pojmenované hodnoty. Jejich jména jsou použitelná jako výrazy, které se vyhodnotí s hodnotou odpovídající proměnné. Deklarace mohou samy

o sobě tvořit **příkaz deklarace**, mohou být použity jako deklarace parametrů funkce, nebo mohou být použity v rámci definice (deklarace s inicializací). Poslední dva případy budou podrobněji popsány dále. Ve všech třech případech se vytváří nová proměnná. V případě příkazu delkarace, je tato proměnná tzv. **viditelná** (tedy její jméno je použitelné jako výraz) ve všech příkazech v bloku po tomto příkazu. Block vytváří tzv. tzv. **scope**, což je část programu, která určuje oblast přístupnosti proměnných. Jiné příkazy mohou také vytvářet scope, ale jen block poskytuje mechanizmus pro uvedení následujících příkazů po příkazu deklarace, ve kterých deklarované proměnné zůstávají viditelné. Každý vytvořený scope v něm umožňuje znovu deklarovat proměnné se jmény, která již byla ve vnějších scopech použita. Takové proměnné tzv. **zastíní** ty dříve deklarované. To znamená, že dokud jsou později deklarované proměnné přístupné, ty předchozí přístupné nejsou. Není možné znovu deklarovat se stejným jménem v rámci stejného scopu.

Hodnota proměnných deklarovaných příkazem deklarace se implicitně nastaví na tzv. **nulovou hodnotu**. Nulová hodnota typu Bool je false a nulové hodnoty typu Int, resp. Float, je nula odpovídající literálu `0`, resp. `0.0`. Nulovou hodnotou pole je pole s každým prvkem nastaveným na nulovou hodnotu v poli obsaženého typu. Imutabliní proměnné kromě IO typů (tedy funkce, buffery a explicitně konstantní jednoduché typy a pole) nelze deklarovat (bez definice).

Deklarace nikdy nevrací.

2.2.7 Definice

Odpovídající neterminál v gramatice: `node_definition`.

Definice je příkaz složený z deklarací reprezentujích definované proměnné a výrazů reprezentujích jejich **inicializační hodnoty**. Vykonání definice provede deklaraci se stejnou sémantikou jako příkaz deklarace a navíc deklarovaným proměnným nastaví uvedené hodnoty. Definovat lze proměnné všech typů. Inicializační hodnoty se musí shodovat s typem uvedeným v deklaraci. Je tedy možné definovat konstantní hodnotu z nekonstantní, ale není možné definovat pole konstantních hodnot z pole nekonstantních hodnot.

Definice nikdy nevrací.

2.2.8 Přiřazení

Odpovídající neterminál v gramatice: `node_assignment`.

Přiřazení je příkaz složený z **cílu** a **přiřazované hodnoty**. Cíl tvoří jméno nebo subscript (neterminál `node_subscript`). *Synakticky jsou cíle tvořeny libovolným výrazem z kategorie expr_post. Omezení na jména a subscripty je sémantické.* Je také dán n-ticí. Cíl představuje koncept podobný l-hodnotám v C, tedy hodnotám, do kterých lze přiřadit. Cíl však v Cynthu není výrazem. Je použitelný jen v přiřazení. Výrazy reprezentují l-hodnoty v Cynthu nejsou. Nahrazují je právě cíle. Cíl se může syntakticky shodovat s výrazem, ale jeho sémantika je jiná. Není tedy možné cíl někam předat.

Je-li cílem přiřazení (resp. jedním z cílů v n-tici) jméno a proměnná tohoto jména je přístupná, její hodnota (resp. hodnoty v n-tici) je nastavena na přiřazovanou hodnotu (resp. hodnoty v n-tici). Přiřazení do subscriptu bude popsáno dále. Obě tzv. **strany** přiřazení, tedy cíl a hodnota, se musí shodovat v typech i v počtu typů. I v přiřazení jsou n-tice na obou stranách ploché. To znamená, že proměnné v cíli se rozloží na posloupnost 1-tic hodnot nezávisle na původní aritě proměnných. Jednotlivé 1-tice z přiřazované hodnoty se tak v uvedeném pořadí přiřadí do jednotlivích 1-tic z cílů.

Přiřazení nikdy nevrací.

2.2.9 Typové aliasy

Typový alias lze vytvořit příkazem **typové definice**. Typová definice má ekvivalentní sémantiku s hodnotovou definicí přenesenou na typy a jména typů. Terminologie je také ekvivalentní až na to, že se nevyužívá pojem typové proměnné, jelikož jde spíše o neměnné typové konstanty. Používá se pojem **jméno typu**, nebo **pojmenovaný typ**. Pro pojmenované typy platí ekvivalentní pravidla scopu.

V aktuální verzi je jen strukturální (nikoliv nominální) typový systém. Nelze tedy vytvořit nový typ, který se nebude shodovat s některým z již existujících typů.

Definice typu nikdy nevrací.

2.2.10 Funkce

Funkce lze vytvořit výrazem **anonymní funkce** (neboli **literál funkce**), nebo příkazem **definice funkce**. Definice funkce je jen zkratka za definici proměnné typu funkce.

Při definici funkce se uvádí výstupní typy, deklarace vstupních **parametrů** a **tělo** funkce. Funkce bez výstupních typů se nazývají **procedury**.

Funkci lze **aplikovat** na dané hodnoty (**argumenty**). Ekvivalentně lze říci, že se funkce **volá** s danými argumenty. Argumenty jsou dány explicitně uzávorkovánou n-ticí. Výstupní typy, výstupní parametry i argumenty mohou být dány prázdným typem, prázdnou deklarací a prázdnou hodnotou. Aplikace funkce není koncipována jako aplikace jedné funkce na seznam argumentů, ale jako aplikace n-tice na n-tici s tím, že syntakticky je ta druhá n-tice explicitně uzávorkována.

Tělo je vždy dáno výrazem. To není nijak omezující, protože block je výrazem. To umožňuje využití již popsané sémantiky blocků a vracení i v tomto kontextu. Funkce je tedy jen zobrazení z argumentů na výslednou hodnotu (se side effecty). Narozdíl od C není block ve funkci sémanticky odlišný od jiných blocků. Funkce reprezentuje parametrizovanou (jejími parametry) část programu (danou jejím tělem), která se při aplikaci vyhodnotí a výsledek vyhodnocení se použije jako výsledek vyhodnocení této aplikace. Před vyhodnocením těla se vytvoří scope pro parametry funkce. V tomto scopu se proměnné dané parametry definují na hodnoty dané argumenty aplikace. Následně se vytvoří další scope pro těla funkce. V těle funkce je tedy možné zastínit parametry.

Funkce mohou jako argument přijmout všechny typy až na funkce. Předávání funkcí jako argumentů je ale v plánu pro další verze.

Výsledek fukce se označuje jako **vrácená hodnota**. Nejde však o pojem návratové hodnoty příkazu. Příkaz definice funkce, ani výraz anonymní funkce či aplikace funkce nemají návratovou hodnotu. Vrácení z funkce nespadá pod pojem předávání. Nevstahují se na něj pravidla o hodnotové a referenční sémantice hodnot. Vrácení nestatických typů probíhá kopií (i u referenčních polí). Vrácení statických typů probíhá referencí.

V těle funkce se mohou nacházet **volné proměnné**. Proměnné jsou volné, pokud nebyly deklarovány (ani definovány) v dané funkci. Proměnné odpovídající této definici mimo funkce (v outermost scopu) jsou vždy sémantickou chybou. Proto je nezvažuji a jako volné proměnné neoznačuji. Funkce se na místě definice pokusí tzv. **zachytit** své volné proměnné z vnějších scopů. Pokud tyto proměnné nejsou deklarovány ani ve vnějším scopu, jde o chybu. Takto zachycené proměnné se označují jako **closure** (nebo **zachycený kontext**). *Na začátku vývoje jsem používal pojem zachycený kontext, proto ho zde zmiňuju, jelikož se může stále vyskytovat v implementaci.* Zachycení proměnných, stejně jako vracení z funkce, nespadá pod pojem předávání. Zachycení nestatických typů probíhá kopií (i u referenčních polí). Zachycení statických typů probíhá referencí.

Odpovídající neterminály v gramatice jsou: `node_function`, `node_function_def` a `node_application`.

2.2.11 Životnost hodnot

Zatímco pojem **viditelnosti** se vztahuje na proměnné, životnost popsisuje vlastnost hodnot, které ani nemusí být přiřazeny do proměnných. Životnost určuje, po jakou část programu hodnota existuje. Výrazy hodnotových typů (jednoduchých typů a funkcí) vytváří vždy nové hodnoty (nově vytvořené či zkopírované), a tím začíná jejich životnost. Životnost pak končí po použití hodnoty tohoto výrazu, jelikož je vždy zkopírována do nové hodnoty. Životnost hodnotových typů přiřazených do proměnných končí s viditelností této proměnné. Životnost referenčních typů je o něco komplexnější. Životnost samotných referencí se řídí stejnými pravidly jako hodnotové typy, ale odkazované hodnoty mají životnost o něco rozšířenou.

Odkazované hodnoty pole se při vytvoření alokují v rámci funkce nebo v nejvnějším blocku (pokud vzniká mimo tělo funkce). To znamená, že odkazované hodnoty existují až do konce vyhodnocování funkce nebo programu. To umožňuje vracení polí referencí. Ne však z funkce. Proto se pole z funkce vrací hodnotou. Odkazované hodnoty statických typů se alokují globálně, tedy v rámci nejvnějšího scopu, a staticky, tedy jedna deklarace vytváří jednu hodnotu, i když je opakovaně spouštěna ve funkci. To umožňuje plně referenční sémantiku statických typů nejen při předávání, ale i při zachycování a vracení z funkcí.

2.2.12 Literály

Literály jednoduchých typů jsou popsány již v syntaktické sekci. Výraz některého z literálů vytváří jednoduchou hodnotu daného typu. Typy těchto hodnot nejsou konstantní.

Odpovídající neterminály v gramatice jsou: `node_bool`, `node_int` a `node_float`.

2.2.13 Literály pole

Literál pole je tvořen seznamem prvků. Prvky pole mohou být dány libovolným výrazem nebo jedním ze tří speciálních prvků: **rozmezí** (**range**), **rozmezí s krokem** (**step range**) a **rozpětí** (**spread**). Prvek dadný výrazem musí být 1-tice (alespoň v aktuální verzi) jednoduchého typu a určuje jeden prvek pole. Prvek daný rozmezím s krokem je tvořen 1-ticemi typu Int nebo Float označujícími **počáteční hodnotu**, **horní** nebo **dolní mez** a **krok**, a určuje prvky pole dané aritmetickou posloupností s diferencí danou krokem, začínající počáteční hodnotou a nepřesahujeící horní nebo dolní mez. Krok může být kladný i záporný. Mez tomu musí odpovídat. Rozmezí bez kroku je ekvivalentní rozmezí s krokem 1 nebo -1 podle dané meze. *V aktuální verzi jsou rozmezí implementována jen s čísly známými za překladu.* Prvek spread je dán jiným polem a určuje prvky zkopírované z tohoto pole. Prvky v literálu se v daném pořadí naskládají do výsledného pole. Pokud se všechny výsledné prvky v literálu shodují v typu, tento typ bez omezení mutability je použit jako typ prvků výsledného pole.

Takto vytvořená pole mají mutabilní prvky i referenci. Pro přiřazení do pole s konstantními prvky, je třeba provést explicitní konverzi. Hodnoty se alokují při vytvoření pole. Následně se jen předává reference. *Implementačně mají nově vytvořená pole skrytou imutabilitu, která se odstraní při prvním použití v explicitně nekonstantním typu. Toto umožňuje optimalizace kompilačních konstant i přes to, že z pohledu uživatele je pole zpočátku nekonstantní a k žádné kopii při konverzi nedochází.*

Literálům odpovídají následující neterminály: `node_array`, `node_range_to`, `node_range_to_by`, `node_spread`.

2.2.14 Subscript

Subscript je výraz nebo cíl daný **kontejnerem** a **lokacemi**. Kontejnerem může být pole, buffer, nebo IO typy, tedy libovolná referenční hodnota. Lokace se uvádí stejným způsobem, jako literál pole. Výše popsané prvky polí jsou určeny i pro použití jako v lokacích, ale v aktuální verzi jsou podporovány jen dva speciální případy lokací: dané jednou hodnotou nebo žádnou hodnotou. Lokace daná jednou hodnotou určuje jeden prvek kontejneru v případě pole nebo bufferu na pozici dané toutou hodnotou. Lokace daná žádnou hodnotou určuje všechny hodnoty v kontejneru a je použitelná nejen pro pole a buffery, ale i pro IO typy pro přístup k odkazované hodnotě. Lokace daná dvěma a více hodnotami (zatím neimplementováno) vybírá nějakou souvislou podmnožinu prvků.

Subscript pole jako výraz se vyhodnotí jako jednotlivý prvek pole nebo kopie celého pole, nebo v budoucích verzích i jako souvislé podpole. Subscript bufferu

jako výraz se vyhodnotí jako prvek bufferu, nebo nové pole zkopírované z hodnot bufferu. Subscript input typu jako výraz se vyhodnotí na odkazovanou hodnotu. Subscript output jako výraz typu je chybou. Z output typů nelze číst.

Subscript pole jako cíl umožňuje přiřazení do jednotlivých prvků pole, nebo do všech prvků naráz (z jiného pole). Subscript bufferu jako cíl je chybou. Do bufferu nelze zapisovat explicitně. Subscript input typu jako cíl je chybou. Do input typů nelze zapisovat. Subscript output typu jako cíl umožňuje zápis do odkazované hodnoty.

2.2.15 Řídící struktury

V Cynthu jsou implementovány násnledující řídící struktury: podmínka **if..else** (zkráceně jen **if**), podmínka **when**, smyčka (loop) **while** a smyčka **for..in** (zkráceně jen **for**). Z toho if a for mohou být výrazy i příkazy (podobně jako block), zatímco ostatní jsou jen příkazy. Všechny tyto řídící struktury obsahují nějaké větve dané výrazy nebo příkazy. If a for mohou být výrazy pouze pokud jejich větve jsou výrazy, jinak (s alespoň jednou větví danou příkazem) jsou if a for příkazy. Ostatní řídící struktury mají větve dané příkazy. Všechny tyto struktury vytváří nový scope pro každou ze svých větví. *V případě smyček je evivalentní označení „větev“ i „tělo“.*

If..else má dvě větve: pozitivní a negativní. Navíc má podmínku danou výrazem typu Bool (1-tice). Je-li podmínka pravdivá, vyhodnotí (resp. vykoná, pokud je if příkazový) se pozitivní větev, jinak se vyhodnotí (resp. vykoná) ta negativní. When má jen jednu větev a je sémanticky ekvivalentní příkazovému if..else s negativní větví danou prázdným blockem. *Takový odlišný konstrukt pro podmínku bez negativní větve zjendnodušuje gramatiku, jelikož se eliminuje problém dangling else.* Pokud je if výrazový, vyhodnotí se na hodnotu vybrané větve. Pokud je příkazový, vrací pokud vrací i jeho vybraná větev.

While má jednu větev a navíc také podmínku danou výrazem typu Bool (1-tice). Svoji větev vykonává dokud podmínka platí. Podmínka se znovu vyhodnocuje před každým vykonáním větve.

For..in má také jednu větev, která se opakuje. Místo podmínky má ale seznam tzv. **range deklarací**. Range deklarace je deklarace proměnné (v aktuální verzi jen jedné proměnné s 1-ticovým typem) s uvedením pole, které určuje hodnoty, co se mají postupně do této proměnné přiřazovat. Všechna uvedená pole musí být stejné velikosti. Nemusí být stejného typu. Větev for loopu se vykoná pro každou n-tici hodnot na sejných pozicích v daných polích. Tedy jde o ekvivalent funkce **map** ve funkcionálním programování. Pokud je for výrazový, vyhodnotí se na pole sestavené z hodnot po každém vyhodnocení větve. Pokud je příkazový, vrací ve chvíli kdy vrací i jeho vybraná větev. For, komě scopu pro větev, vytváří nad ním i scope pro range deklarace. Range deklarace tedy lze zastínit. Lze je i mutovat (pokud nejsou označeny jako konstantní). Neovlivní to ale pole z range deklarace. Definice proměnných z range deklarací je také předávání, a tedy se prvky přiřazeují hodnotou. *Pole hodnot známých za překladu se v mnohých případech optimalizují na aritmetickou posloupnost, pokud nějaké odpovídá. Iterace se pak překládá jak klasický for loop s inkrementováním hodnot, spíše než průchod*

alokovaného pole.

2.2.16 Ostatní operace

Pro běžné operace jsou implementovány vestavěné operátory. Podobají se operátorům v C syntkticky a částečně i sémanticky, ale u některých z nich jsou značné rozdíly. Precednece všech operátorů je stejná jako v C. Uvedu všechny vestavěné operátory v tabulkách (2.1, 2.2 a 2.3) s jejich názvy, příjímanými typy operandů, syntaxí danou příkladem výrazu a sémantikou danou odpovídajícím výrazem v C. Oprátory jsou polymorfické v tom smyslu, že akceptují různé typy. Typy všech operandů ale musí být vždy shodné mezi sebou a musí se shodovat s jedním z uvedených typů v tabulce.

Syntaktickým kategoriím operátorů odpovídají neterminály: `expr_or`, `expr_and`, až `expr_pre` (v přiložené gramatice uvedeny v tomto pořadí).

Aritmetické operátory

Název	Výraz	Typy	Sémantika
plus	`+a`	Int, Float	`a`
mínus	`-a`	Int, Float	`-a`
sčítání	`a + b`	Int, Float	`a + b`
odečítání	`a - b`	Int, Float	`a - b`
násobení	`a * b`	Int, Float	`a * b`
dělení	`a / b`	Int	`(a - 2 * (a < 0) + 2 * (b < 0)) / b`
dělení	`a / b`	Float	`a / b`
modulo	`a % b`	Int	`((a % b) + b) % b`
modulo	`a % b`	Float	`return fmod((fmod(a, b) + b), b)`
umocnění	`a ** b`	Float	`fpow(a, b)`

Tabulka 2.1: Aritmetické operátory

Celočíselné dělení zaokrouhluje dolů narozdíl od C, které zaokrouhluje směrem k nule. Podle toho se řídí i celočíselné a reálné modulo. Taková sémantika dělení a hlavně modula je vhodnější k použití s cyklickými buffery s negativními indexy. *Aktuálně jsou reálná čísla implementována pomocí typu float v compile-time C++ interpreteru i run-time C kódu. Tedy u modula a umocnění jde přesněji o funkce fmodf a fpowf.* Celočíselné umocnění je v plánu pro další verze, ale v této verzi to nebylo prioritou.

Logické operátory

Název	Výraz	Typy	Sémantika
negace	`!a`	Bool	`!a`
konjunkce	`a && b`	Bool	`a && b`
disjunkce	`a || b`	Bool	`a || b`

Tabulka 2.2: Logické operátory

Logická konjunkce i disjunkce jsou tzv. **short-circuiting**, to znamená, že druhý operand se nevyhodnotí, pokud je výsledek jednoznačný i bez něj. *Při optimalizacích vyhodnocujících části kódu za překladu se tyto operátory takto nechovají, ale nemá to vliv na sémantiku, jelikož za překladu se vyhodnocují a vykonávají jen části programu, které nemají side effecty.*

Porovnávací operátory

Název	Výraz	Typy	Sémantika
rovnost	`a == b`	Bool, Int, Float	`a == b`
nerovnost	`a != b`	Bool, Int, Float	`a != b`
ostré nerovnosti	`a < b a > b`	Bool, Int, Float	`a < b a > b`
neostré nerovnosti	`a <= b a >= b`	Bool, Int, Float	`a <= b a >= b`

Tabulka 2.3: Porovnávací operátory

Hodnoty typu Bool jsou uspořádány tak, jak jsou uspořádány odpovídající hodnoty po konverzi na typ Int, tedy `false < true`.

2.2.17 Konverze

Implicitní konverze nikde neprobíhají. Typy se musí při jakémkoliv předávání shodovat. K explicitní konverzi je určen výraz konverze, který je dán výsledným typem a hodnotou ke konverzi. Konverze jsou možné jen mezi některými typy. Konverze spadá pod předávání.

Odpovídající neterminál v gramatice: `node_conversion`.

Všechny jednoduché typy jsou na sebe konvertovatelné. Mají při tom sémantiku ekvivalentní odpovídajímu castu v C, až na konverzi z Floatu do Intu, která narozdíl od C provádí zaokrouhlení dolů. Konstantnost se může přidávat i odebírat, jelikož jde o kopie.

Pole lze v aktuální verzi konvertovat jen na pole obsahující hodnoty shodných typů. Konstantnost reference se může přidávat i odebírat (jde o kopii reference), zatímco konstantnost obsahovaných hodnot se může jen přidávat. Konverze na menší velikost je možná, na větší nikoliv. Při konverzi se předá reference. Tedy konverze na konstantní obasžené hodnoty je vpořádku, ale naopak by znamenalo mutabilitu původně konstantních hodnot. Stejně tak konverze na menší velikost je vpořádku, jde jen o **pohled** nad existujícím polem, který může ignorovat zbylé alokované hodnoty.

Input typy lze konvertovat na typ jejich obsažené hodnoty. Jde o ekvivalentní operaci subscriptu s prázdnou lokací.

Buffery lze konvertovat na menší buffery. Jde o stejný princip jako u polí. Konverze z bufferu na pole aktuálně není implementována, ale lze toho dosáhnout pomocí subscriptu s prázdnou lokací a další případné konverze velikosti.

Funkce lze konvertovat na buffery, což je popsáno v další kapitole.

Ostatní konverze nejsou možné.

2.2.18 Buffery

Buffer velikosti n reprezentuje historii poslndních n vzorků nějakého signálu. Z bufferů lze číst subscriptem s nekladným indexem. Nula označuje poslední vzorek signálu v historii (ten nejnovější) a čísla směrem dolů od nuly označují starší vzorky.

Do bufferů nelze zapisovat explicitně. Pro zápis do bufferů jsou použity tzv. **generátory**. Generátory jsou funkce typu `Float (Int)` nebo `Float ()`, které reprezentují signál, tedy funkci vracející vzorek typu Float a případne přijímající parametr typu Int označující čas daný počtem uběhlých vzorků od začátku běhu syntezátoru. Takto definované funkce lze konvertovat na buffer. Tato konverze má side effect takový, že alokuje nový buffer (staticky) a zaregistruje k němu odpovídající generátor.

Za běhu se pak zapisují ve smyčce sekvenčně nové hodnoty do všech bufferů a pro to se musí vypočítat hodnoty odpovídajících generátorů. V kompilační konfiguraci (`inc/config.hpp`) je uvedena vzorkovací frekvence. Zápis do bufferů se opakuje právě v této frekvenci (pokud to definovaný Cynth program stíhá). Tato vzorkovací frekvence je také přístupná uvnitř Cynth kódu jako proměnná `srate`.

2.2.19 Kompilační kostanty

Literály jsou výrazy, hodnota kterých je známa za překladu. Výrazy neobsahující žádné proměnné mají také hodnotu známou za překladu. Deklarace konstantní proměnné (konstantní jednoduché typy, pole s konstantní referencí i hodnotami a funkce) z hodnoty známé za překladu tvoří proměnnou s hodnotou známou za překladu. Deklarace nekonstantní proměnné (nebo přiřazení do proměnné) ztrácí hodnotu známou za překladu. Výrazy, resp. příkazy, obsahující jen literály a proměnné s hodnotou známou za překladu mají výslednou hodnotu, resp. návratovou hodnotu, známou za překladu. Hodnoty známé za překladu se označují jako **kompilační konstanty**. Kompilační konstanty lze použít pro určení velikosti typu pole nebo bufferu. Implementačně se provádí různé optimalizace díky kompilačním kostantám, takže je vhodné co nejvíce proměnných deklarovat jako konstantní. Je důležité podotknout, že v příkazech či výrazech obsahujících alespoň nějakou, i nepoužitou, nekonstantní proměnnou se ztrácí kompilační hodnota. Tedy např. aby funkce mohla být vyhodnocena striktně za překleadu, musí mít i konstantní parametry.

3. Implementace

V této kapitole popíšu přehled základních postupů i hlubších detailů implementace.

3.1 Jazyk a nástroje

Překladač Cynthu je napsán v C++20. Využívají se jen obecně přístupné vlastnosti a standardní klihovny standardu C++20. Překlad je otestován na překladačích GCC 10, Clang 10 a 11. Poslední verze MSVC by měla také podporovat vše relevantní z C++20, ale není to otestováno. Části programu překladače jsou generovány nástroji Bison (konkrétně verze 3.7.1) a Flex (verze 2.6.1). K generaci kódu a překladu kompletního programu jsou připraveny Makefily. Makefile pro překlad kompletního programu navíc spoléhá na soubory se závislostmi mezi implementacemi a headery vygenererované pomocí GCC. Překladač Cynthu vygeneruje kód v jazyce C (konkrétně v dialektu gnu17), který je následně potřeba přeložit do spustitelného souboru. Dialekt gnu17 podporuje jak GCC tak Clang.

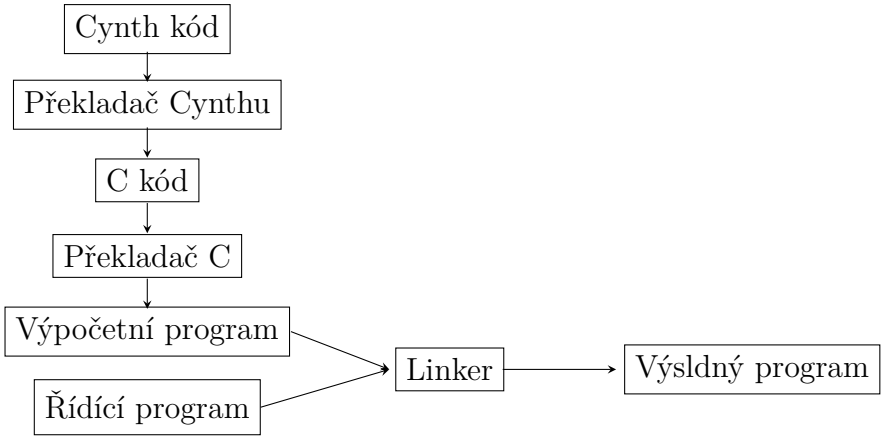
3.2 Platforma

Program dle původního rozsahu cílil na platformu Windows a zvukové karty podporující technologii ASIO. Jelikož je aktuální verze programu omezená jen na překlad základní emulace bez propojení se zvukovou kartou, tato omezení odpadají. Výsledný C program je multiplatformní až na jedno systémové volání. Kvůli němu je třeba v kompilační konfiguraci `inc/config.hpp` nastavit cílovou platformu, podle čehož se ve výsledném programu vygeneruje `#include` headeru `<windows.h>` nebo `<unistd.h>`.

Překlad je otestován na Windows v prostředí MSYS2 s MinGW-v64 a ve WSL (Windows Subsystem for Linux) s překladači GCC a Clang.

3.3 Kompletní postup sestavení

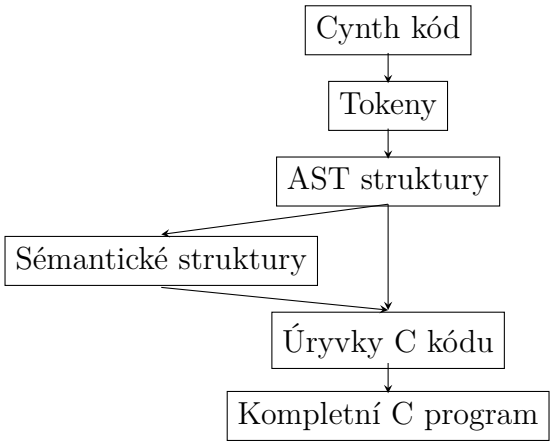
Překladač Cynthu dostane na vstupu **Cynth kód** a na výstupu vydá **C kód**. Tento kód je dále třeba přeložit **překladačem pro C**. Takto přeložený **výpočetní program** je určený ke statickému slinkování s předkomilovaným **řídícím programem**, který má na starosti komunikaci se zvukovou kartou, vykreslení a reakci na GUI a reakci na MIDI vstup, do **výsledného programu**. *Tento řídící program není součástí této verze.*



Mezikrok s překladem do C umožňuje využít následující optimalizace překladače C. Úroveň optimalizací překladače C lze nastavit v konfiguraci makefilu `config.mk`.

3.4 Fáze překladu

Samotný překlad ze Cynthu do C je rozdělen na několik fází: Lexikální analýza rozdělí vstupní text na tokeny. Syntaktická anlýza dle vstupních tokenů vytvoří AST (Abstract Syntax Tree) reprezentované v C++ strukturách. AST struktury se dále mohou transformovat do pomocných sémantických struktur, nebo rovnou překládat do úryvků C kódu. Sémantické struktury se buď využívají na validaci částí programu za překladu, nebo přímo vyhodnocování částí programu za překladu. Některé sémantické struktury se dále přeloží do úryvků C kódu. Nakonec se všechny vytvořené úryvky C kódu sestaví do výsledného C programu. Tento program je obsažen v jediném `.c` souboru.



Pro další slinkování s řídícím programem bude třeba vytvořit i header soubor (`.h`) s interfacem, skrze který by řídící program obdržel informaci o vygenerovaných statických datech (IO typy a buffery) a mohl by do nich zapisovat nebo číst.

3.5 Struktura implementace

3.5.1 Interface AST a sémantických struktur

Pomocné struktury mají pro výše popsané přechody určený přesný interface. Všechny struktury jsou v C++ implementovány jako obyčejné structy a polymorfismu je docíleno pomocí `std::variant`ů. Každý takový struct implementuje potřebnou funkcionalitu jako metodu. Tyto metody se nevolají přímo, ale přes daný soubor volných funkcí, představující interface různých typů struktur, které se aplikajují přes `std::visit`. *Přesněji řečeno je implementována utilita, která umožňuje „lift“ funkcí nad různé typy, mezi něž patří i variant s implementací aplikace funkce visitor patternem.*

Přechod z AST struktury do sémantické struktury označuji jako **resolution**, přechod do sémantické struktury nebo C kódu (v závislosti na možnostech dané AST struktury) jako **processing** a přechod ze sémantických struktur do C kódU jako **translation**. Snažil jsem se držet jednotného přístupu, ale časem se ukázalo, že jsou potřeba stále nové konstrukty. Ve výsledku interface není tak jednoduchý. Kromě uvedených případů se také řeší: Získávání hodnot kompilačních konstant, kontrola shody typů a extrakce volných proměnných. Dále je funkcionalita deklarací a definicí rozdělena mezi struktury příslušných typů. Pseudokódem, jde zhruba o následující interface:

```
processStatement    (Node) -> ()
processExpression   (Node) ->  (Value | TypedExpr)
processArrayElement (Node) -> [(Value | TypedExpr)]

resolveType               (Node) -> Type
resolveDeclaration        (Node) -> Declaration
resolveRangetDeclaration (Node) -> RangeDeclaration
resolveTarget             (Node) -> Target

translateValue (Value) -> TypedExpr
translateType  (Type)  -> Name

valueType (Value)       -> Type
get       (Value) <Out> -> Out

sameTypes  (Type, Type) -> Bool

processDefinition  (Type, ResolvedValue) -> ()
processDeclaration (Type)                -> ()

extractNames     (Node) -> [Name]
extractTypeNames (Node) -> [Name]
```

Tyto funkce se spouštějí rekurzivně od kořene programu (nejvnějšího bloku). Pro kompletní překlad je aktuálně potřeba až 3 průchody AST. Jeden průchod vyřeší základní validaci i překlad zároveň. Další dva průchody jsou potřeba na extrakci volných jmen proměnných a jmen typů. Bylo by ale možné je spojit do jednoho průchodu. Spojit extrakci jmen i překlad by ale mohle být nevýhodné. Aktuálně

totiž překladač ignoruje mnohé části kódu, které se nikdy nespustí a provede na nich jen extrakci, což lehce urychluje překlad.

3.5.2 Kontext

Dále je velmi podstatnou částí překladových algoritmů tzv. **context** (neboli **kontext**). Kontext je soubor několika struktur, které řídí data sdílená mezi rekurzivními voláními výše uvedených funkcí. **Global context** obsahuje globální úryvky C kódu (globální alokace dat, definice typů a funkcí, includy standardních headerů), globální počítadlo pro unikátní identifikátory jmen a informace o generátorech. Vytvoří se jednou na začátku překladu a používá se po celou dobu běhu překladu. **Function context** obsahuje úryvky C kódu pro alokace na úrovni funkce, a také aktuálně deklarované parametry odpovídající funkce. Vytvoří se jednou při každé deklaraci (nebo compile-time evaluaci) funkce. Po dokončení deklakace či evaluace se odstraní. **Branching context** obsahuje aktuální číslo větve pro komplexnější případy vracení. Vytvoří se jednou při zpracování výrazových blocků a následně se také odstraní. **Lookup context** obsahuje definice proměnných (a také typů), jak jsou deklarované a viditelné z pohledu Cynth kódu, a k nim buď kompilační konstantu, nebo název odpovídající proměnné v C kódu.

Nakonec **main context** spojuje tyto kontexty dohromady, a také obsahuje úryvky C kódu pro lokální příkazy. Main context obsahuje odkazy na global context, function context a branching context a pak přímo obsahuje lookup context. Main context spolu se svým lookup contextem odpovídá scopům v Cynth kódu. Pro nový scope se vytvoří dceřinný main context i s novým, odpovídajícím lookup contextem. Ostatní kontexty jsou tvořeny mimo main context dle potřeby.

3.5.3 Výsledný výpočetní program

Tyto kontexty shromažďují úryvky C kódu roztříděné na různé kategorie (typy, funkce, atd), které se na konci složí do výsledného programu. Výsledný program je složen do následující struktury:

```
// Dependencies:
#include <<header>>
...

// Types:
struct cth_<name> {
    <definition>
};
...

// Static data:
<type> cth_<name>;
...

// Functions:
<out> cth_<name> (<in>) {
    <definition>
```

```
}
...

// Signal handler:
volatile bool cth_stop = false;
void cth_stop_handler (int _) {
    cth_stop = true;
}

int main () {
    // Initialization:
    {
        // Outermost scope user code:
        ...

    }
    ret:
    printf("Synthesizer initialized.\n");

    // Main loop:
    double sample_rate = 44.100000; // kHz
    int time           = 0;
    double tick_time   = 1 / sample_rate; // ms

    signal(SIGINT, cth_stop_handler);

    while (!cth_stop) {
        clock_t start, end;

        start = clock();

        // Write:
        <buffer>.data[(<buffer>.offset + 1) % 1] =
            <generator>(<closure>, <arg>, ...).e0;
        ...

        // Step:
        <buffer>.offset = (<buffer>.offset + 1) % 1;
        ...

        ++time;

        end = clock();

        double elapsed =
            ((double) (end - start)) / CLOCKS_PER_SEC * 1000; // ms
        double remains = tick_time - elapsed;
        if (remains > 0) {
            sleep(remains / 1000);
        }
    }
```

```
    // Debug:
    printf("Buffer ‘<buffer>‘ stopped at %i.\n", <buffer>.offset);
    for (int i = 0; i < 1; ++i) {
        printf("%f\n", <buffer>.data[i]);
    }

    return 0;

}
```

Závěr

Výsledkem práce je překladač programovacího jazyka určeného pro popis emulace syntezátorů. Výsledný program nebyl napojen na zvukovou kartu ani na MIDI vstup pro otestování reálného výsledku real-time výpočtů signálů, takže nebyla otestována výsledná latence. Byla ale s úspěchem otestována rychlost překladu, schopnost vypočítávat potřebné vzorky signálu v rozumném rozsahu, a také expresivita jazyka k popisu základních operací při zpracování signálů a k vytváření komponent syntezátoru.

Výsledek práce sice neodpovídá plnému rozsahu zadání, ale i to bylo pro mě osobně velkým výsledkem. Pro dokončení této práce jsem nastudoval tři nová témata, se kterými jsem měl naprosto minimální zkušenosti, a to zpracování zvukových signálů, návrh programovacího jazyka a implementace překladače. Komě získaných teoretických i praktických zkušeností je hlavním výslekdem práce právě zjištění, nakolik komplexní je vytvořit i takto jednoduchý programovací jazyk bez předchozích zkušeností.

Seznam tabulek

A. Přílohy

A.1 Cynth.zip

Zip archiv obsahující kompletní zdrojový kód programu a základní dokumentaci. Části programu, jako vygenerovaný parser a lexer nebo závislosti pro Makefile, jsou přítomny v archivu a není třeba je opět sestavovat. Žádný spustitelný program přítomen není. Výsledný překladač i výsledné přeložené emulace syntezátorů je třeba sestavit pomocí Makefilu.

Archiv je nahrán prostřednictvím SISu. Navíc ale ještě uvedu odkaz na osobní repozitář na GitHubu: `https://github.com/egst/cynth`.