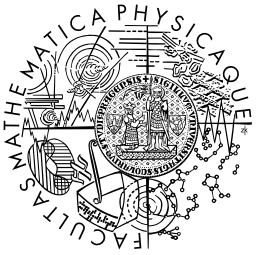

MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

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

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

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

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. Program je cílený na platformu Windows a zvukové karty podporující technologii ASIO.

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 singnals 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. The program's target platform is Windows and sound card's supporting the ASIO technology.

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. Specifikace

1.1 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.

1.1.1 Tokeny

Mezi tokeny patří několik tzv. **klíčových slov** (neboly **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. (Bude reprezentovat rekurzivní 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.

TODO: Escaping dollar signs.

TODO: Hyphenation.

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. Ani 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.*

1.1.2 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**. Přikazy 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 dvé 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**.

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

To samé platí pro n-tice **typů**, n-tice deklarací a n-tice **cílů**.

```
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
```

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.

```
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
```

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`).

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

TODO: More notes about the syntax?

1.2 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. Výrazy se vyhodnocují. Vyhodnocený výraz má vždy tzv. **výslednou hodnotu**, Pojmy výraz a hodnota jsou záměnné. (Výraz je pojem syntaktický, zatímco hodnota sémantický.) zatímco 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 neformálními 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.

1.2.1 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.

1.2.2 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 vyrvořit **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 vyrvořit **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. (V aktuální verzi.) 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 jsou funkce. Funkce jsou hodnoty a z pohledu uživatele se předávají hodnotou. 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, za-

tímco 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.

1.2.3 Definice

1.2.4 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.

1.2.5 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.*

1.2.6 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 přikazů 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.

1.2.7 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.

1.2.8 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) Přiřazení do subscriptu budo popsáno dále. Obě tzv. **strany** přiřazení (tedy cíl a hodnota) se musí shodnovat 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.

1.2.9 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.

1.2.10 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`.

1.2.11 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 skopírované) 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 pra-

vidly 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.

1.2.12 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`.

1.2.13 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.*

Odpovídající neterminály v gramatice jsou: `node_array`, `node_range_to`, `node_range_to_by`, `node_spread`.

1.2.14 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 dané dvěma a více hodnotami (zatím neimplementováno) vybíra nějakou souvislou podmnožinu prvků.

Subscript pole jako výraz se vyhodnotí jako jednotlivý prvek pole, nebo kopie celého pole. (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.

1.2.15 Řídící struktury

V Cynthu jsou implementovány 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 (podobě 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.

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 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 ale má 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, ve chvíli kdy vrací 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.*

1.2.16 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 (1.1, **??** a **??**) s jejich názvy, přijímanými 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 a musí se shodovat s jedním z uvedených typů v tabulce.

Syntaktickým kategoriím operátorů odpovídají neterminály v gramatice: `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 1.1: 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 verzí 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 1.2: 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`
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 1.3: Porovnávací operátory

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

1.2.17 Konverze

Implicitní konverze nikde neprobíhají. Typy se musí při jakém koliv 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.

1.2.18 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 ma 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`.

1.2.19 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.

Závěr

Seznam tabulek

A. Přílohy

A.1 První příloha