TideScript: A Domain Specific Language for Peptide Chemistry

Nicholas Morris
University of Glasgow
Glasgow, United Kingdom
2885282M@student.gla.ac.uk

Blair Archibald
University of Glasgow
Glasgow, United Kingdom
blair.archibald@glasgow.ac.uk

S. Hessam M. Mehr
University of Glasgow
Glasgow, United Kingdom
hessam.mehr@glasgow.ac.uk

Abstract

Synthetic peptides are becoming increasingly important in industry and academia, including applications in peptide-based therapeutics. While Solid Phase Peptide Synthesis (SPPS) is largely automated, current software options for describing this process are instrument-specific, with no unifying standard protocol specification. Post-synthesis modification of peptides in the solution phase is also performed manually. As a result, discovery and process optimisation remains an iterative and manual process, requiring large amounts of human intervention and labour.

Here we introduce TideScript: a domain specific language for peptide chemistry. TideScript aims to provide a programming language to standardize protocol descriptions, allowing unambiguous specification, increased automation, and autonomous optimization. At present, TideScript is a formal specification language, with scope to expand into full execution on hardware.

We give a full description of TideScript, including its syntax and a small-step operational semantics, and show how TideScript can encode three common peptide protocols: peptide chain assembly by SPPS, peptide cleavage and thiol-maleimide conjugation. We compare this approach to existing non-peptide-specific chemical protocol languages, χDL and BioScript.

CCS Concepts: • **Applied computing** → *Chemistry*; • **Software and its engineering** → **Domain specific languages**.

Keywords: Peptide Chemistry, Protocol Languages

ACM Reference Format:
Nicholas Morris, Blair Archibald, and S. Hessam M. Mehr. 2025. TideScript: A Domain Specific Language for Peptide Chemistry. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3759429.3762627

1 Introduction

Chemistry research has a reproducibility problem [8]. Experimental protocols are expressed in continuous prose and use imprecise terminology—*e.g.* "add **dropwise**" or "stir **overnight** at **room temperature**"—making it difficult to re-run experiments manually, and almost impossible to automate their execution. This reproducibility problem can be tackled through the development of domain-specific (protocol) languages (DSLs). DSLs for chemistry promise to standardise procedures across different instruments and platforms; increase reliability; facilitate automation; and enable autonomous process optimisation [15].

While laboratory automation is growing, current work is often focused on specific tasks within a larger overall workflow and only considers specific proprietary hardware. Recently, there has been progress in programmable chemistry across industry and academia, and many syntheses have been carried out [16, 19].

Peptide chemistry is a chemistry sub-field with critical applications in the pharmaceutical industry. Over 80 peptide therapeutics are approved globally [35], including ozempic (semaglutide) for treating obesity and diabetes [20]; enfuvirtide to treat HIV [5]; and ziconotide, to treat chronic pain [10]. Peptides can also be conjugated (chemically bonded) to small drug molecules to increase their selectivity and efficacy. Peptide therapeutics can also be used to prime the immune system for cancer treatment [34].

Automated execution of peptide chemistry procedures entails unique considerations and challenges. Here we introduce TideScript, a domain specific protocol language for *peptide chemistry*. TideScript has primitives for execution of basic chemical steps, *e.g.* mixing, agitation, heating, and domain modelling of the underlying chemistry. We give a formal syntax and operational semantics and show how TideScript can be used to encode several existing peptide protocols. Currently, TideScript provides formal specification of peptide protocols, but we aim to develop it to a fully executable on automated laboratory hardware. TideScript has been designed to maintain *portability*, that is to say it can be developed to target a variety of hardware architectures.

TideScript also has scope for development of a richer type system, one that can catch procedural and chemical errors. We plan to develop typing that increases safety, reliability and reproducibility of peptide chemistry. TideScript initially focuses on *peptide* chemistry, in order to focus in and better capture the problem domain, but there is scope for expansion into broader chemistry.

"The purified peptide was taken up in DMF (approximately 2 mg/mL). An aliquot of a stock solution of 17[anti-bacterial derivative] in DMF (2.87 mM) was added such that the volume added corresponded to 3 eq, and the reaction mixture stirred at RT for 3 h."

(A)

```
#declare peptide
peptide pep_a = <SADC(acm)LKSC(acm)LRMCPDRIIL>;

#declare molecule
molecule mol_b = (C14H31N7O8S);

#declare solvent
solvent dmf;

#declare solutions by passing in materials
solution sol_x = [pep_a(2 mgMl) in dmf];

solution sol_y = [mol_b(2.87 mM) in dmf];

#define protocol
protocol <solution> conjugate
(pep_sol mol_sol volume unit) = {
    new_solution = mix pep_sol(1 eq)
    + mol_sol(3eq) at volume unit;
    changeTemp new_solution to 21 C;
    agitate new_solution;
    wait 3 hours;
    return new_solution
}

#call protocol at different scales
call conjugate (sol_x sol_y 1 mL);

call conjugate (sol_x sol_y 100 mL);

call conjugate (sol_x sol_y 1 L);
```

(B)

Figure 1. (A) The published literature procedure for a peptide thiol-maleimide from reference [30] and (B) The same protocol expressed in TideScript.

The necessity for such a language is shown in the example of Figure 1 (procedure taken from [30]) showing a protocol where a peptide and anti-bacterial molecule are bound together *via* a thiol-maleimide conjugation procedure. Figure 1A shows the plain text protocol description from the original publication. While all the steps are well-specified, it is not clear how to automate this procedure. Using TideScript, we can instead express the protocol unambiguously as shown in Figure 1B. While slightly longer, we are much more explicit about which peptides, molecules, solvents, and solutions are involved and how they should be combined. Ambiguity is removed, *e.g.* "room temperature" is defined explicitly as 21 °C.

This protocol does not specify a volume of solution but instead instructs the chemist to add 3 equivalents, meaning it is applicable to *any scale*. To allow scalable protocols in TideScript, *protocol* constructs operate similarly to functions in computer programming languages: *i.e.* they are parametrised by specific values. In this case, we define conjugate over different volume units, and show how we can call it at multiple scales. Chemical reactions do not necessarily scale linearly, that is to say different outcomes can be observed at different scales due to a variety of factors [25]. This is something to consider when automating protocols at different scales, and new kind of reaction optimisations may be required. What was successful at a smaller scale may not be at larger scales.

We were motivated to enable the encoding of similar protocols in TideScript, and make the following research contributions:

- We design TideScript: a novel, high-level protocol language, for peptide synthesis.
- We present formal syntax and operational semantics for the language.
- We show how TideScript is expressive enough to capture three existing protocols, and compare the approach with existing chemical protocol languages χDL and BioScript.

2 Peptide Chemistry

Peptides and proteins are chains of amino acids. Peptides are small proteins or sections of proteins. The point at which a growing peptide chain becomes a protein is not explicitly defined, but generally proteins are much larger (greater than 50 amino acids in length) and tend to adopt more complex three dimensional structures [13].

Peptides and proteins are ubiquitous in nature and mediate a large variety of biological functions. This makes them ideal to produce therapeutic effects. Peptides can can be expressed from cells, derived from animal tissue, or produced fully synthetically [13]. Synthetic peptides provide great flexibility as modifications can be made at specific amino acid sites to expand their functionality [35].

Peptides are generally either modified while dissolved in a solution or *synthesised* via binding to a solid support. To fully capture peptide chemistry, TideScript supports both mechanisms.

2.1 Solution Phase Modifications

In-solution peptide modifications involve protocols where peptides are dissolved in solutions and reactions are performed on them. The procedure outlined in figure 1 is a thiol-maleimide conjugation, but there are many other examples of in-solution modifications [35].

There are also many other factors to consider and optimise for, such as the reaction conditions (temperature, time etc.),

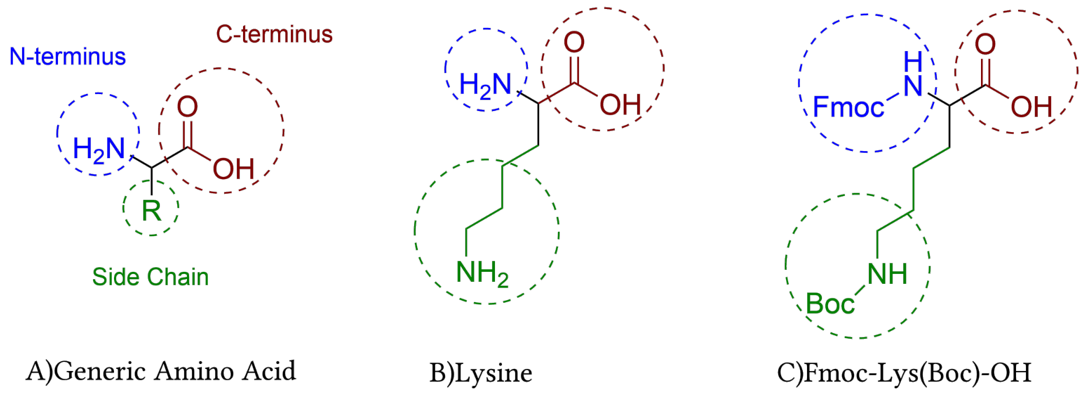

Figure 2. A) The structure of a generic amino acid. B) An example amino acid, Lysine. C) Corresponding SPPS building block, Fmoc-Lys(Boc)-OH, with the appropriate protecting groups on the N-terminus and side chain.

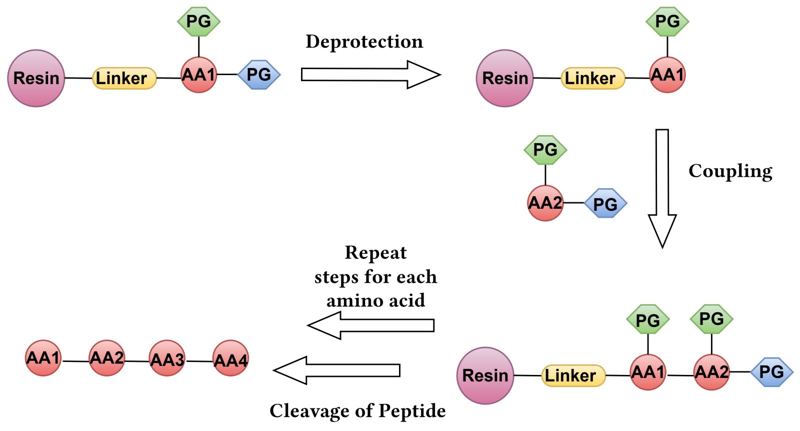

Figure 3. Typical workflow of peptide assembly by Solid Phase Peptide Synthesis (SPPS).

and whether the appropriate amino acid sites are present (*e.g.* thiol-maleimide conjugation requires a cysteine residue, but multiple cysteine residues may require a protecting group strategy).

2.2 Solid Phase Peptide Synthesis

Before in-solution modifications take place, peptide chains themselves are usually assembled using a technique known as solid phase peptide synthesis (SPPS). Here amino acids are joined together (coupled) into a growing peptide chain that is attached to a solid support. The solid support retains the growing peptide chain and allows for unreacted starting materials and side products to be washed away [11]. This process is amenable to automation, and has been automated for many decades. However, removal of the peptide chain from the solid support (cleavage), and the in-solution modifications described before, largely remain manual processes [11].

An amino acid has 2-3 reactive functional groups, depending on the side chain. In order to assemble the peptide sequence in the desired way, we need to have *protecting groups* on the functional sites we do not want to react. The protecting groups are removed in a specific order, to allow assembly at the correct orientation. Compare the structure of the amino acid Lysine, and the structure of a corresponding SPPS building block, Fmoc-Lys(Boc)-OH, see figure 2.

When an amino acid is coupled in the chain, the N-terminal protecting group is removed, allowing the next amino acid to be coupled. The side chain protecting groups remain intact until the peptide is fully assembled, then they are removed as the peptide is removed from the solid support, see figure 3. The important thing to remember is that is peptide chains are assembled in a sequential building block manner, through a series of successive steps.

3 `TideScript`: A Protocol Language for Peptide Chemistry

`TideScript` is designed with careful consideration of the aspects of the problem domain and how these need to be captured and modelled within the constructs of the language. To mimic the nature of existing chemical process definitions, we use a declarative/imperative approach[1] where series of statements give precise instructions to equipment in the system architecture. These actions have an effect on the *state* of the program. Rather than *computational* state, the *state* refers to the status of the experimental apparatus (the presence of materials, their temperature, whether they are agitating etc.). The system architecture, refers to the set up of equipment (pipettes, electrodes etc.) rather than a computer chip architecture.

Language statements fall into two categories: *declaration* and *operation*. Declaration statements declare the materials involved in the experiment, *e.g.* a particular solution such as 0.1 g/mL NaCl in water, while operation statements describe actions performed on them.

We give the abstract syntax in figure 4, and specific protocol examples are in section 7.1.

3.1 Materials and Declaration Statements

Solution phase peptide reactions involve the manipulation and interactions of solutions, and the chemical reactions that occur within these solutions. A solution consists of a (possibly empty) set of solutes (peptides or molecules), dissolved at particular concentrations, in a set of at least one solvent. Solutes and solvents can be thought of as primitives and solutions as composite data structures. Programs begin with users declaring these primitives.

3.1.1 Molecules. Molecule declarations associated identifiers, *e.g.* a molecule name, to a molecule description. The description of the molecule is either its molecular formula or its SMILES specification and this can be used to enable chemistry specific modelling. For example, deriving molecular weights and using them in mass spectrum analysis. SMILES is the simplified molecular input line entry system (SMILES). It is a specification describing full chemical structures as ASCII strings, and it is used commonly in computational chemistry [37]. `TideScript` uses the open-source OpenSMILES specification [33]. SMILES specifications are more powerful than simple formulas as they elucidate the full structure of the compound to aid in more comprehensive analysis, reaction monitoring and process optimisation [28].

3.1.2 Peptides. We introduce constructs for peptide declarations. While peptides are essentially just large molecules, they are a separate construct in `TideScript`, as peptide chemistry is chiefly concerned with modifications to these special peptide molecules. Peptide declarations also involve assigning an identifier to a description. Like a molecule, they can be described by their formula or SMILES string. However, it is intended that they are usually described by their sequence of amino acids, denoted by their corresponding one letter code, *e.g.* PEPTIDE. The compiler can derive the formula and SMILES representation of a linear peptide from its peptide sequence.

[1]An open question is whether it is worth breaking from this tradition and discovering if chemistry could be better specified through other approaches, *e.g.* logic languages.

3.1.3 Solutes. Solutes describe entities that can be dissolved in solvents. In `TideScript`, a solute is simply either a peptide or molecule.

3.1.4 Solvents. Solvents are media (usually liquids) that are used to dissolved solutes. Common examples include water, DMF (dimethylformamide, not to be confused with digital microfluidics), DMSO etc. The user can select a solvent from a standard library, and each solvent type encodes record type information that is relevant to experiments, namely: polarity, whether or not it is protic (proton donating) and its refractive index. These data are relevant for future type-checking of reactions and downstream analysis. For example, a particularly hydrophobic peptide will not be soluble in a polar solvent, or a reaction might only proceed with a protic solvent.

The user can define their own solvents if they are not available in the standard library. The minimum requirement is a string that describes the solvent, and additional information, *e.g.* on polarity, is optional (at the cost of weaker chemical analysis).

3.1.5 Solutions. A solution consists of one or more solutes, dissolved at a particular concentration, in one or more solvents. For example, a salt solution consisting of sodium chloride dissolved in water at 1 g/mL. Solutions are represented in the language as data structures with an assigned identifier.

When Solutions are declared, they are located, as opposed to solutes and solvents, when they are instantiated they are a physical entity that *operations* can be performed on. That is, they will map to physical locations within a particular hardware setup: a specific reaction vessel, or cell in a DMF setup.

There are two important features of solutions. First, when they are instantiated, they are assumed to have infinite volume. That is they can continually be dispensed from without incurring any errors from the compiler. This allows "running" the protocol and letting the program *inform* the user how much of a particular solution is required at setup phase. Secondly, solutions are assumed to be within their own container, there is not a separate construct for containers (*e.g.* flasks), *i.e.* a user instantiates a solution x, solution x is contained within its own container. This improves readability, but more importantly keeps the language hardware agnostic, as the domain of reaction vessels in a Chemputer vs a digital microfluidic board is very different.

3.1.6 Neat Solutions. Many chemical compounds involved in experiments may exist as neat liquids, there is a *neat* construct where the user can declare a solution that is simply the neat form of a previously declared molecule.

```
molecule methanol = (CH3OH);
methanol_sol = neat(methanol);
```

3.1.7 Units. There is support for different units when declaring solutions. This is because reactions are performed at a variety of scales: from microgram to kilogram scales. Units within the language allow for more readable and expressive programs. Concentrations can also be expressed in moles per unit volume as well as a mass (*e.g.* grams) per unit volume. We support automatic inference between grams and moles based on the compounds molecular weight.

3.1.8 Example. Declaration statements may look like this:

```
solvent dmso;
molecule a = (CH2O);
molecule b = (CC(=O)Cc1ccccc1);
peptide pep1 = MKLWGSAPFT;
solution x = [a(1 mgML) in dmso]
```

3.2 Operation Statements

The other category of statements within the language are *operation* statements. These will involve actuating components of the system architecture, and making changes to the state of the program. Operations statements can further be divided into operations that instantiate new solutions, and operations that change the status of existing solutions

3.2.1 Operations to Instantiate New Solutions. The language has constructs for *dispensing* and *mixing*, these statements will generate a new solution in the program state. For example, dispensing 5 mL of solution x leads to a new solution x' with identical composition(in terms of solvents, dissolved solutes, and their respective concentrations). Adding instead, involves dispensing from two existing solutions x and y and creating a new solution z that has both sets of solutes and solvents concatenated.

```
solution x = mix a(1 eq) +  b(2 eq) at 5 mL
```

3.2.2 Operations That Change the Status of a Solution. There are also operation that will change the status of a solution. This includes changing its agitation state (*e.g.* is it being stirred), or applying *changeTemp* to the solution.

```
changeTemp  a at 21 C;
agitate a;
```

3.2.3 Protocols. Protocols are analogous to functions from traditional programming languages. This is a powerful construct that allows the user to define a series of operations, that can be executed on input parameters. Higher order protocols are not allowed, *e.g.* one protocol cannot be used as a parameter to another. The user can however call protocols within the body of the protocol, allowing modularisation and building larger protocols (as demonstrated in section 7).

The return type of a protocol can either be void—*i.e.* nothing is returned—or it can return a new solution, when the protocol creates a new solution from *e.g.* dispensing or mixing.

```
protocol<void> shake_solution (sol){
    agitate sol;
    wait 20 s;
    agitate sol
 }

protocol<solution>(sol_1 sol_2 volume){
    sol = mix sol_1(1 eq) + sol_2(2 eq) at volume
}
```

3.3 Timing

As many experimental procedures require leaving the experiments for a certain amount of time. *e.g.* "stir for 3 hours", we introduce a *wait* construct. The user can specify an amount of time to wait, with support for units seconds, minutes, hours. Currently, only the front-end compiler is implemented, so the waiting times are not yet actionable and are simply just encoded in the script syntax.

```
wait 40 s;
wait 5 mins;
wait 4 hours;
```

4 Syntax

The abstract syntax of `TideScript` is outlined in figure 4. A Program, P consists of a zero or more statements. In `TideScript`, a statement can consists of either a declaration, an operation or a protocol definition. While it is common to have all declarations up-front, there is no strict requirement for this, and `TideScript` allows mixing declarations and operations. Declaration statements describe the presence of a material in an experiment by assigning an identifier, v, to them. Solution declarations have an identifier, v_1, a set of identifiers, v_2 (which are references to previously declared peptides or molecules), with their corresponding concentrations, and a list of solvents that they are dissolved in within the solution.

Operation statements describe physical operations carried out by the hardware on the experiment materials. Agitate calls a reference of a previously declared solution. Heat is applied to a reference identifier at a particular temperature. Dispense dispenses a volume, vol, from from a reference solution. Mix dispenses two volumes from two solutions and combines them.

Protocol definitions use the terminal protocol, with an identifier for the protocol name, a return type t, a list of identifiers as input parameters, and a nested series of statements that are to be executed on the input parameters.

Finally, Call executes a protocol by passing in parameters (by reference).

```
<Statement> ::= <Declaration> | <Operation> | <Protocol>
<Program> ::= <Stat>*
<Type> ::= Peptide | Molecule | Solvent | Solution | Void
<Sequence> ::= [A-Z]+
<Formula> ::= [A-Z][0-9][a-z]?
<Smiles> ::= ...
<Declaration> ::= <PeptideDeclaration> | <MoleculeDeclaration> | <SolventDeclaration> | <SolutionDeclaration> |
<NeatDeclaration>
<PeptideDeclaration> ::= peptide v = <Sequence>
<MoleculeDeclaration> ::= molecule v = [<Formula> | <Smiles>]
<SolventDeclaration> ::= solvent v
<SolutionDeclaration> ::= solution v = [ v_1(<conc>) ]* v_2* <vol>
<NeatDeclaration> ::= solution v' = neat(v)
<Operation> ::= <Agitate> | <Heat> | <Dispense> | <Mix> | <Call>
<Agitate> ::= agitate v
<ChangeTemp> ::= changeTemp v <float>
<Dispense> ::= v' =dispense v <vol> <ConcUnit>
<ConcUnit> ::= C | K <Vol> ::= <float> <VolUnit>
<VolUnit> ::= μL | mL | L
<Conc> ::= <float> <ConcUnit>
<ConcUnit> ::= mg/mL | mol/L
<Wait> ::= wait <float> <TimeUnit>
<TimeUnit> ::= s | mins | hours
<Mix> ::= v' = mix v_1 vol_1 v_2 vol_2 | v' = mix v_1(<float> eq) v_2(<float> eq) at <Vol>
<Protocol> ::= protocol v (<Type>) v_params* [<Statement>]*
<Call> ::= call v (v_args)*
```

Figure 4. Abstract Syntax for `TideScript`. The full specification for smiles is omitted and available elsewhere [33]. Keywords are highlighted in bold, units in blue and identifiers in green.

5 Semantics

5.1 Definitions and Notation

Consider each operation, where P denotes the program and S denotes the program state.

$$\langle P, S \rangle \tag{1}$$

The state of the program includes sets of all declared materials: This is peptides (Pep), molecules (Mol), solvents ($Solv$), solutions (Sol), and Protocols ($Prot$). We use lower case letters for elements of a set.

$$mol \in Mol \quad \text{(a molecule within the program set)} \tag{2}$$

$$pep \in Pep \quad \text{(a peptide within a set)} \tag{3}$$

$$solvn \in Solvn \quad \text{(a solvent within a set)} \tag{4}$$

$$sol \in Sol \quad \text{(a solution within a set)} \tag{5}$$

$$prot \in Prot \quad \text{(a protocol within a set)} \tag{6}$$

So a state s, can be described as

$$s = \langle Pep, Mol, Solv, Sol, Prot \rangle \tag{7}$$

As we care, in particular, about the manipulation of solutions this is the most interesting set. A solution contains a set of dissolved peptides or molecules, $[pep|mol]$, and their corresponding concentration, conc. There is also a set of solvents contained within the Solution. Additionally, there are other properties of the solution such as the volume, vol, temperature, T, and the boolean of its agitating state, ag.

$$Sol = \langle [pep|mol]^*, Solvn^*, vol, T, ag \rangle \tag{8}$$

When we want to describe an update to the program state, with a new component (such as a solution, protocol etc), with the identifier, v, we will deconstruct the state and show the component as being in the union of one of the state sub-sets. *e.g.* $S(v \cup Sol)$, but we will omit the entire tuple.

5.2 Declaration Rules and Axioms

The [PeptideDeclaration] rule describes declaring a peptide in the program set, the identifier, v, is passed in by reference and must not exist already in the set of peptides in the program state, Pep. The declared peptide is now part of the program state. The same logic applies to the [MoleculeDeclaration] and [SolventDeclaration] rules.[SolutionDeclaration] is more complicated, the solution identifier, v_1 must not be declared already in the program set, the set of peptides or molecules, v_2^* must be declared already, as well as the set of solvents, v_3^*. The Solution, identified by v is then appended into the program state.

The [ProtocolDeclaration] rule describes defining a new protocol, the protocol identifier must not exist in the program set, and the defined protocol is added to the program set, with a list of parameters, v_2, contained within a list of operations to perform on those parameters.

$$[PeptideDeclaration]\frac{v \notin Pep}{\langle peptide\ v :: P, S\rangle \longrightarrow \langle P, S(v \cup Pep)\rangle}$$

$$[MoleculeDeclaration]\ \frac{v \notin Mol}{\langle molecule\ v :: P, S\rangle \longrightarrow \langle P, S(v \cup Mol)\rangle}$$

$$[SolventDeclaration]\frac{v \notin Solvn}{\langle solvent\ v :: P, S\rangle \longrightarrow \langle P, S(v \cup Solvn)\rangle}$$

$$[SolutionDeclaration]\ \frac{v_1 \notin Sol \quad v_2^* \in S(Pep, Mol) \quad v_3 \in S(solvn)}{\langle solution\ v_1\ [v_2, conc]^*\ v_3^*\ Vol :: P, S\rangle \longrightarrow \langle P, S(v_1 \cup Sol)\rangle}$$

$$[ProtocolDeclaration]\frac{v \notin Prot}{\langle protocol\ v\ t\ v_2^*\ (D|O)^* :: P, S\rangle \longrightarrow \langle P, S(v \cup Prot)\rangle}$$

Figure 5. Small-step operational semantic rules for declaration statements in `TideScript`.

5.3 Operation Rules and Axioms

The [DispenseOperation] rule describes dispensing a volume n, from a solution with the identifier,v, where the volume of the solution is greater than the dispensation volume. Solution v now has a volume reduced by n and $v^{'}$ is added to the set with a volume of n.

The [DispenseAllOperation] rule describes dispensing a volume vol, from a solution v, where the all of the solution is being dispensed. The solution v is removed from the program set and a new solution, $v^{'}$ is added to the set with a volume of n.

The [Agitate] rule toggles the agitation state of the solution. Agitation can take many forms depending on the equipment. This could be shaking, stirring, rotating etc. Additionally, it is not just a binary toggle, as it can have magnitude, *e.g.* "stir at 200 rpm". The semantics currently model agitation as a toggle, as this suffices for many protocols, that will simply state "stir for x hours". We do however aim to expand upon this in future.

The [ChangeTempOperation] rule alters the temperature of an existing solution to a temperature value, n, within the program set. Semantically, the program assumes we have full control over the solution temperature, this is not the case in practice as it can be difficult to precisely control a solution's temperature, one can only apply heat or cooling. Section 9.2 discusses in more detail how equipment *conformance* may be tackled in the future.

The [MixOperation] rule combines the dispense rules for two solution, v_1 and v_2, and adds a new solution, $v^{'}$ to the program set and the parent solutions have their volumes reduced accordingly.

Additionally, the solutes in the new solution $v^{'}$ will have a new concentration, based on their initial concentrations from the parent solution and a dilution factor, q, based on the final volume.

$$q = \frac{V_{parent}}{V_{total}} \tag{9}$$

Where V_{parent} is the volume dispensed from the parent solution and V_{total} is the total volume of the new solution, $v^{'}$. The new solution concatenates the solutes, solutes' respective concentrations (adjust with dilution factor) and the solvents from the parent solutions.

Where A is a solute, that is either a peptide or molecule and C is their respective concentrations.

$$\begin{aligned} v^{'} = \langle A = A_1 :: A_2, \quad C = qC_1 :: qC_2, \\ Solv^{'} = Solv_1 :: Solv_2, \quad V = vol_1 + vol_2, \quad T\rangle \end{aligned} \tag{10}$$

The [MixOperationAllSolution1], [MixOperationAllSolution2], [MixOperationAll] rules describe the scenario where all of v_1 is dispensed, all of v_2 is dispensed, and all of both solutions are dispensed, respectively.

The [CallProtocol] rule describes calling a previously defined protocol. This follows call-by-reference semantics and every protocol must have an explicit return statement (but this might be the void solution). A protocol has a set of params, and a sub program, Pr. When called with a set of arguments, we map these into the existing program state by creating references, *i.e.* $p \mapsto a$ means to create an reference/alias that allows p to be used in place of a. This creates a new state, $S^{'}$. The protocol (sub-program) runs until it hits a Return statement and this is added back to the program state. Since we only support call-by-reference, the remaining program continues with state S rather than any protocol-modified state S'', *i.e.* protocol specific solutions etc. do not escape the protocol scope.

The program halts when an attempt is made to execute a program that does not conform to the semantic rules.

6 Extending `TideScript` to Support Solid Phase Peptide Synthesis (SPPS)

The existing version of `TideScript` is expressive enough to handle in-solution peptide modifications. To support solid phase peptide synthesis, we introduce additional language constructs.

[*DispenseOperation*]

$$\frac{v \in Sol \quad v_{vol} > n}{\langle Dispense\ v\ n :: P, S\rangle \longrightarrow \langle P, S(v(vol - n)v'(vol = n) \cup Sol)}$$

[*DispenseAllOperation*]

$$\frac{v \in Sol \quad v_{vol} = n}{\langle Dispense\ v\ n :: P, S\rangle \longrightarrow \langle P, S(v')(vol = n) \in Sol, v \notin Sol)\rangle}$$

[*AgitateOperation*]

$$\frac{v \in Sol}{\langle Agitate\ v :: P, S\rangle \longrightarrow \langle P, S(v(v_{agitate} = \neg v_{agitate}) \cup Sol)}$$

[*ChangeTempOperation*]

$$\frac{v \in Sol}{\langle ChangeTemp\ v\ to\ n\ : P, S\rangle \longrightarrow \langle P, S(v(temp = n) \cup Sol)\rangle}$$

[*MixOperation*]

$$\frac{v_1, v_2 \in Sol \quad vol_{v1} > vol1 \quad vol_{v_2} > vol2 \quad v^{'} \notin Sol}{\langle v^{'} \ = mix\ v_1\ vol_1\ nu_2\ vol_2 :: P, S\rangle \longrightarrow \langle P, S(v^{'}(vol = vol_1 + vol_2), v_1(vol - vol_1), v_2(vol - vol_2) \cup Sol)\rangle}$$

[*MixOperationAllSolution1*]

$$\frac{v_1, v_2 \in Sol \quad vol_{v1} = vol1 \quad vol_{v_2} > vol2 \quad v^{'} \notin Sol}{\langle v^{'} \ = mix\ v_1\ vol_1\ nu_2\ vol_2 :: P, S\rangle \longrightarrow \langle P, S(v^{'}(vol = vol_1 + vol_2), nu_2(vol - vol_2) \cup Sol)\rangle}$$

[*MixOperationAllSolution2*]

$$\frac{v_1, v_2 \in Sol \quad vol_{v1} > vol1 \quad vol_{v_2} = vol2 \quad v^{'} \notin Sol}{\langle v^{'} \ = mix\ v_1\ vol_1\ nu_2\ vol_2 :: P, S\rangle \longrightarrow \langle P, S(v^{'}(vol = vol_1 + vol_2), nu_1(vol - vol_1) \cup Sol)\rangle}$$

$$[MixOperationAll]\ \frac{v_1, v_2 \in Sol \quad vol_{v1} = vol1 \quad vol_{v_2} = vol2 \quad v^{'} \notin Sol}{\langle v^{'} = mix\ v_1\ vol_1\ nu_2\ vol_2 :: P, S\rangle \longrightarrow \langle P, S(v^{'}(vol = vol_1 + vol_2) \cup Sol)\rangle}$$

[*CallProtocol*]

$$\frac{v = \langle prm_0, \ldots, prm_n, Pr\rangle \in Prot \quad \substack{arg_0 \ldots arg_n \in S \\ prm_0 \ldots prm_n \notin S} \quad S^{'} = S \cup (prm_0 \mapsto arg_0, \ldots, prm_n \mapsto arg_n) \quad \langle Pr, S^{'}\rangle \longrightarrow \langle Return\ t, S^{''}\rangle}{\langle r = Call\ v(arg_0 \ldots arg_n) :: P, S\rangle \longrightarrow \langle P, S \cup r\rangle}$$

Figure 6. Small-step operational semantic rules for operation statements in TideScript.

6.1 Resins

In SPPS peptides are assembled on a solid support that retains the growing peptide chain as side products and unreacted materials are washed away. As a central concept, we add these as a new type of entity within our language (alongside the usual peptides, molecules, solvents and solutions).

Resins have a loading value, in mmol/g to describe the number of active sites, that allow binding to a peptide chain, per gram of resin material. Each resin therefore tracks a mass in g, a loading value in mmol/g, and an associated peptide chain.

6.1.1 Reaction Vessels. In TideScript, solutions, and only solutions, are assumed to map to a physical location, *e.g.* a vessel, DMF cell etc. In SPPS, solutions are always explicitly added and drained in reaction vessels (RVs), with the growing the peptide chain linked to the resin. Therefore, it is necessary to instantiate an RV to perform SPPS operations. An RV will have a resin, and a set of (possibly empty) solutions associated with it.

6.1.2 Amino Acid Solutions. In SPPS, *coupling cycles* are defined in order to couple each amino acid in the sequence. The coupling cycle is often the same, but will call just a different amino acid building block solution. Users may want to iterate through a peptide sequence, *e.g.* MKLWGSAPFT, performing the same coupling cycle. In order to do this, solutions need to be associated with the amino acid building block they contain, denoted by their corresponding one letter code. Declaring an amino acid solution (aa_solution) is the same as a regular solution, except there is inclusion of the one letter code.

```
solution ala_solution =[ala(0.14 M) in DMF, A]
```

Users can then call the for sequence construct, to iterate through a sequence and call a protocol, using the appropriate amino acid solutions each time.

```
forSequence MKLWGSAPFT {
   call coupling_cycle(rv aa_solution)
 }
```

6.1.3 Repetition Statements.

SPPS also involves many repeated steps in cycles, thus it is useful to introduce *repetition* to the language, where a user declares a statement is to be executed multiple times.

6.2 Additional Syntax for SPPS

Figure 7 outlines the additional syntax required to encode SPSS. This is an *extension* of the original abstract syntax, *i.e.* Declarations include solutes, peptides etc. *and* new declarations. Additional declaration statements allow instantiating resins and RVs, and new operations include adding and draining solutions from RVs, and agitating RVs (which is used to react the solution to cause binding). When an RV is instantiated its solvent set is empty.

There are also constructs for declaring amino acid solutions, with a corresponding one letter code, aa. There are only 20 naturally occurring amino acids, not one for each letter in the alphabet, but there is scope for the user to include non-standard amino acid building blocks. There is also the ForSequence construct. Note $\langle Sequence \rangle$ was defined in the initial semantics.

```
<Declaration>::= <Resin> | <RV> | <AASolution>
<Resin> ::= resin v(<Mass> <Loading> v_peptide)
<RV> ::= RV v v_resin
<AASolution> ::= solution v = [ v_1(<conc>) ]* v_2*
<vol> <AA>
<Mass> ::= <float> <MassUnit>
<MassUnit> ::= g | mg | μg
<Loading> ::= <float> mmol/g
<AA> ::= [A-Z]
<Operation> ::= <AddSolutiontoRV> | <DrainRV> |
<AgitateRV>
<AddSolutiontoRV> ::= add v_solution(<Vol>) to v_rv
<AgitateRV> ::= agitate v
<DrainRv> ::= drain v
<Repetition> ::= <Statement> * <int>
```

Figure 7. Extended syntax required to incorporate SPPS into TideScript. Keywords are highlighted in bold, units in blue and identifiers in green.

6.3 Semantics

The program state now includes a set of resins, Resin, a set of RVs, RV, and a set of amino acid solutions, Sol_{aa}.

$$resin \in Resin \quad \text{(a resin within the state set)} \tag{11}$$

$$rv \in RV \quad \text{(an RV within a set)} \tag{12}$$

$$sol_{aa} \in Sol_{aa} \quad \text{(an amino acid solution within a set)} \tag{13}$$

The [ResinDeclaration] rule declares a resin, with an identifier, v, a mass, loading value and peptide chain. The declared resin is added to the program state and the peptide chain, v_{pep} must previously be declared in the program.

The [RVDeclaration] rule declares an RV with identifier v, containing a previously declared resin, v_{resin}.

The [AgitateRV] rules toggles the agitation state of the rv with identifier v.

The [AASolutionDeclaration] rule describes the declaration of an amino acid solution, it is the same as a [Solution-Declaration] except it must not be part of the amino acid solution set ($Sol_a a$) and has a corresponding one letter code, denoted aa.

The [AddSolutiontoRV] dispenses a volume n, from solution v_{sol}, where the volume of the solution is greater than the dispensing volume. The program state is updated so the dispensed solution, v'_{sol} is part of the union of solutions within the rv, $Sol_r v$, and the volume of the parent solution is reduced by n.

The [AddAllSoltutionToRV] occurs when all of the solution is added to the rv. In this case the original solution is removed from the program state.

The [DrainRV] rule drains all solutions from the rv, the set of solutions within the rv, $Sol_n u$ is now nil and they are removed from the program state.

The [Repetition] rule allows an operation statement to be executed multiple times. The Operation statement (and this only makes sense for operations and not, for example, declarations), O, is executed n times; where n is a natural number and is greater than zero. This is essentially a program transform (and could be implemented as such) but as we have not defined other transforms we choose to express it as an operational rule here.

The [ForSequence] rule iterates through each one letter code in the passed-in sequence, and call the passed in protocol with each corresponding amino acid solution.

7 Expressivity

7.1 Expression of Real-World Literature Procedures

To show `TideScript` is capable of modelling realistic protocols, we show how to encode 2 further existing, textually defined, protocols, in addition to the conjugation procedure outlined in figure 1. We use these protocols to compare `TideScript` with other chemical programming languages.

7.1.1 Peptide Sequence Assembly.

Peptide assembly using SPPS utilises successive coupling for each amino acid in the desired sequence. These coupling cycles can further be broken down into deprotection and coupling cycles (see figure 3).

A specific Fmoc deprotection (removal) cycle, and an amino acid coupling cycle are described in [30], outlined in figure 9.

We now express this in `TideScript`. We first declare the materials used in the experiment (using our declaration operations): the reaction vessels (and the nature of the resin contained within) and the solutions (including their concentrations).

This protocol uses reaction vessels with 100 mgs of NovaSyn TGT resin, preloaded (at 0.2 mmol/g) with a protected

$$[ResinDeclaration]\ \frac{v \notin Resin \quad v_{pep} \in Pep}{\langle resin\ v\ (mass, v_{pep}, loading) :: P, S\rangle \longrightarrow \langle P, v \cup S\rangle}$$

$$[RVDeclaration]\ \frac{v \notin RV \quad v_{resin} \in Resin}{\langle rv\ v(v_{resin}) : P, S\rangle \longrightarrow \langle P, v \cup S\rangle}$$

$$[AgitateRV]\ \frac{v \in RV \quad v(agitate) = false}{\langle Agitate\ v :: P, S\rangle \longrightarrow \langle P, v(agitate \neg v_{agitate}) \cup S\rangle}$$

$$[DrainRV]\ \frac{v \in RV}{\langle drain\ v :: P, S\rangle \longrightarrow \langle P, v(Sol_{nu} = \epsilon) \cup S\rangle}$$

$$[AASolutionDeclaration]\ \frac{v_1 \notin Sol_{aa} \quad v_2^* \in S(Pep, Mol) \quad v_3 \in S(solvn)}{\langle solution\ v_1\ [v_2, conc]^*\ v_3^*\ Vol, aa :: P, S\rangle \longrightarrow \langle P, S(v_1 \cup Sol)\rangle}$$

$$[AddSolutionToRV]\ \frac{v_{sol}, v_{rv} \in Sol, RV \quad v_{sol}(vol > n)}{\langle add\ v_{sol}\ n\ v_{rv} :: P, S\rangle \longrightarrow \langle P, v_{sol}(vol - n), v_{rv}(v_{sol} \cup Sol_{rv}) \cup S\rangle}$$

$$[AddAllSolutionToRV]\ \frac{v_{sol}, v_{rv} \in Sol, RV \quad v_{sol}(vol = n)}{\langle add\ v_{sol}\ n\ v_{rv} :: P, S\rangle \longrightarrow \langle P, S(v_{sol} \notin Sol, v_{rv}(v_{sol} \cup Sol_{rv}) \cup S\rangle}$$

$$[Repetition]\ \frac{O \text{ is a single operation} \quad n \in \mathbb{N} \quad n > 0}{\left\langle O * n :: P, S\right\rangle \rightarrow \left\langle \underbrace{O :: \cdots :: O}_{n \text{ times}} :: P, S\right\rangle}$$

$$[ForSequence]\ \frac{v_{prot} \in Prot \quad Sequence = \langle aa_0 ... aa_n \in Sol_{aa}\rangle}{\langle forSequence < aa_0, ...aa_n > v_{prot} :: P, S\rangle \longrightarrow \langle Call\ v_{prot}(aa_0) :: ... :: Call\ v_{prot}(aa_n) :: P, S\rangle}$$

Figure 8. Semantic rules to extend `TideScript` to incorporate the expression of solid phase peptide synthesis (SPPS).

> “The reaction syringe containing N-terminal Fmoc-protected peptide was added piperidine in DMF (40% v/v, 1.5 ml). The mixture was agitated for 20 s every min for a total of 3 min. The reagents were removed by filtration under vacuum, and the resin washed with DMF (4 1.5 ml). Piperidine in DMF solution (40% v/v, 0.75 ml) was added to the reaction syringe followed by DMF (0.75 ml) to make an overall 20% v/v solution of piperidine in DMF. This mixture was agitated for 20 s every min for a total of 10 min.”

> “The reaction syringe was added Fmoc-protected amino acid (0.600 ml, 0.140 M in DMF, 4 eq.), HBTU (0.600 ml, 0.140 M in DMF, 4 eq.), and DIPEA (0.300 ml, 0.560 M in DMF, 8 eq.). The mixture was agitated for 20 s every 3 min for a total of 40 min. The reagents were removed by filtration under vacuum, and the resin washed with DMF (4 1.5 ml).”

Figure 9. The literature protocol from reference [30], describing the deprotection cycle (left) and the coupling cycle (right).

amino acid: Asp (single letter code D). In `TideScript`, we instantiate a peptide with the single amino acid to include in the resin declaration:

```
peptide pre_load = <D>
resin resin_1 = {100 mg, 0.2 mmol/g, pre_load}
rv rv_1 = {resin_1}
```

We then declare the rest of the molecules and solvents involved in the experiment. Then we instantiate the solutions containing them with appropriate concentrations (*e.g.* 0.14 M).

Note here we import a library of standard amino acid solutions.

```
import standard_aa_solutions(0.14 M)
solvent dmf;

molecule piperidine = (C5H11N);
solution deblock = [piperidine(0.14 M) in dmf];

molecule hbtu = (C11H16F6N5OP);
solution hbtu_sol = [hbtu(0.14 M) in dmf];

molecule dipea = (C8H19N);
solution dipea_sol = [dipea[0.56 M] in dmf];
```

```

protocol<void> dmf_wash (rv){
    add dmf(1.5 mL) to rv;
    drain rv
}

protocol<void> shake_rv (rv){
    agitate rv;
    wait 20s;
    agitate rv;
    wait 40s
}

protocol<void> shake_rv_3min (rv){
    agitate rv
    wait 20 s;
    agitate rv;
    wait 160 s
}

protocol<void> deprotection (rv){
    add deblock(1.5 mL) to rv;
    call shake_rv (rv) * 3;
    drain rv;
    call dmf_wash (rv) * 4;
    add deblock(0.75 mL) to rv;
    add dmf(0.75 mL) to rv;
    call shake_rv (rv) * 10;
    drain rv;
}

protocol aa_coupling (rv aa_solution){
    add aa_solution(4 eq) to rv(1 eq);
    add hbtu(4 eq) to rv(1 eq) +
    add dipea (8 eq) to rv(1 eq);
    call shake_rv_3min (rv) * 13;
    drain rv;
    call dmf_wash_rv (rv) * 4;
}

protocol<void> coupling_cycle (rv aa_solution){
    call deprotection (rv);
    call aa_coupling (rv aa_solution);
}

protocol spps (rv sequence){
    for sequence {
    rv = coupling_cycle aa_solution
    }
}

call spps (rv_1 SADCLKSCLRMCPD);
```

Figure 10. TideScript code for expressing the SPPS protocols from the published literature descriptions from Figure 9.

We then utilise the protocol definition functionality. Protocols can be used to define washing, deprotection and coupling. These protocols can then be combined into a single coupling cycle, for each amino acid to be coupled.

Finally, we pass in a peptide sequence to a defined protocol, in order to iterate through each amino acid sequence and carry out the coupling cycles. This script is outlined in figure 10.

7.1.2 Peptide Cleavage.

The peptide cleavage procedure from reference [30] is outlined in figure 11.

The cleavage cocktail uses a combination of compounds that are liquid at room temperature, so the cocktail is described as these compounds being combined at a specific ratio. The `TideScript` language constructs allows the user to declare a solution as simply being a neat liquid of a declared molecule. A new solution can then be declared by mixing them. These molecules can be thought of as being dissolved compounds at specific concentrations. The script expressing this procedure is outlined in figure 12.

"The resin was washed with CH2Cl2 (3 3 ml), MeOH (3 3 ml) and Et2O (3 3 ml) and dried (desiccator) followed by adding TFA/EDT/ H2O/TIPS (88:5:4:2; 3 ml) to the reaction syringe. The syringe was then agitated for 3 h at RT. The cleavage cocktail was drained from the vessel under vacuum and Et2O (10–15 ml) added to the filtrate. The resultant precipitate in solution was stored at 20 C for 30 min prior to being spun at 4000 rpm for 10 min at 4 C to produce a crude peptide pellet. The supernatant Et2O was decanted off, and the peptide washed a further three times with Et2O. The crude peptide pellet was then re-dissolved in minimum water and freeze-dried for storage prior to purification."

Figure 11. Cleavage procedure described in reference [30].

This `TideScript` protocol expresses the main elements of the cleavage procedure from figure 11, however it does not capture steps such as freeze-drying as these are primarily manual steps not supported in the language.

7.2 Limitations of Expressivity in `TideScript`

`TideScript` is designed to express common protocol elements from peptide chemistry: movement, mixing, agitation, heating etc. We have shown it to cover solid phase peptide assembly, most of peptide cleavage and conjugation in sections 1, 7.1.1 and 7.1.2. However, as demonstrated in section 7.1.2, protocols sometimes contain special instructions such as freeze-drying, filtering under vacuum or using a centrifuge. These currently require manual intervention as automatic lab hardware is unlikely to support all of these features.

In future, we plan to introduce additional language constructs to allow these features, where, although not fully automated, the user can specify the semantic outcome of an operation (*e.g.* whether a peptide is "used-up" after a step). This will allow protocols to be more complete and also support strong reasoning. For example, manual process blocks could be specified for vacuum filtration:

```
    filtrate = manual "vacuum filter"
    outcome (del peptide p; return peptide "..." )
```

To keep the language contained, and ensure full control over the semantics, we do not add this to the first version of `TideScript`. Many languages include 'escape hatches', which allow users to escape an abstraction layer to reach a lower level [7]. Examples include the Java unsafe API [22],

```
solvent dcm;

solvent methanol;
solvent ether;

molecule tfa = (C2HF3O2);
solution tfa_sol = neat(tfa);

molecule edt = (C2H4S2H2);
solution edt_sol = neat(edt);

molecule water = (H2O);
solution water_sol = neat(water);

molecule TIPS = ([(CH3)2CH]3SiH);
solution TIPS_sol = neat(TIPS);

solution cleavage_cocktail = [tfa(88%) + edt (5%) +
    water(4%) + TIPS(2%) ];

protocol<void> wash (rv solution){
  add solution to rv;
  drain rv;
}

protocol<void> final_wash (rv){
  call wash (rv dcm);
  call wash (rv methanol);
  call wash (rv ether);
  drain rv;
}

call final_wash (rv_1);

protocol<void> cleave (rv){
    add cleavage_cocktail to rv;
    agitate rv;
    wait 3 hours;
}

call cleave(rv_1);
```

Figure 12. TideScript script for expressing the cleavage procedure from figure 11.

and the use of inline assembly in C/C++ [21]. We can employ a similar concept here, allowing us to manually perform specialised actions not yet covered by the language/hardware.

In general, the question of whether a language can capture all of (peptide) chemistry is an interesting one: is there an equivalent to a Turing machine/Turing completeness for chemistry/chemical languages, and is there a hierarchy of chemical language power.

Luca Cardelli discusses how language design choices can require bottom-up design of hardware [6]. This would allow new language constructs to guide future hardware design (a step change from classic computation). While many pieces of lab equipment are computer controlled, automating their integration still remains challenging [6].

This bottom-up design of hardware has been demonstrated by the Cronin group, and the development of their 'Chemputer' platforms, targeted with their χDL language. They have integrated systems of pumps, valves, and flasks to carry out automated chemistry. With language constructs that describe hardware components, and describe operations such as stirring, heating, and moving reagents between said hardware components. We discuss χDL in more detail in section 8.1.

Some examples of the challenges of expressing peptide protocols in TideScript, and constructs that would need to be introduced are outlined below.

7.2.1 Performing Reactions in a Vacuum or Under Inert Atmosphere.

Many chemical reactions, including some performed in peptide synthesis, require an inert atmosphere in TideScript, typically achieved through an atmosphere of nitrogen or argon, or performing the reaction in a vacuum. This is often required as oxygen, or water molecules present in atmospheric air, can interfere or kill certain reactions [24]. There is not currently a notion of atmosphere. If special atmospheres are to be incorporated, semantically they may need to be tracked in the program state. For example, "the current atmosphere = 100% Nitrogen", and there may be purging operation steps that alter this state. The integration of atmosphere tracking in TideScript would greatly expand its expressiveness.

Hardware interfaces to carry this out have already been developed. The Cronin group have developed the "Schlenkputer" to carry out such reactions [4]. "Schlenk" here refers to specialist type of glassware used to facilitate inert reactions. The χDL language has constructs for describing the schlenk hardware, as well as *evacuateandrefill* and *purge steps* [14].

7.2.2 Performing Reactions at Extreme Temperatures.

Similarly, some reactions are required to be performed at very high or very low temperatures. While there is a construct for changing temperature in TideScript, semantically, temperature change is an assignment, and the actual temperature change is less controlled and deterministic than that. A construct for some hardware feedback to confirm desired temperatures before proceeding may be required, and specialist hardware is needed for anything beyond mild heating or cooling. TideScript could benefit from feedback based control, and mechanisms that can manage the difference between real world value and the semantic modelling if they de-sync.

7.2.3 Gas-Phase Reactions.

TideScript has constructs for solid and solution phase reactions, but sometimes gaseous compounds are used in chemical synthesis [3]. Constructs for gaseous reagents will need to be incorporated in order to express and execute these protocols. This also ties in with the notion of atmosphere mentioned earlier.

7.2.4 Purifications.

We have also not addressed purifications in this work. There are many purification techniques in chemistry but peptide products are most commonly purified by reversed phase HPLC [9]. The details of HPLC are beyond the scope of this paper, but there exist many instruments to automate this process. Nevertheless, expression of purification protocols would aid in the development of integrated and reproducible workflows.

8 (Bio)Chemistry Protocol Languages

8.1 χDL

χDL is chemical programming language, for automating general synthetic chemistry protocols. The researchers have developed a Chemputer architecture, which is a modular robotic platform for carrying out operations such as the reactions, filtration, separation and evaporation [32].

The higher level language compiles down into low-level instructions for robotic execution. This allows the same synthesis files to operate on different hardware platforms, providing the necessary resources are present [27, 32].

While the language initially targeted their own Chemputer architecture, they have expanded on the compiler to be hardware-agnostic, and demonstrated the operation of χDL scripts on different Chemputer platforms, and OpenTrons robotics [29].

A construct of χDL has been developed, called blueprints. Reaction blueprints are analogous to functions in traditional programming languages, similar to protocols in `TideScript`. The concept is to allow users to define sets of operation into irreproducible, parallelised workflows, that can be reused on different reagents and conditions [39].

8.1.1 Comparing χDL to `TideScript`.

`TideScript` is not the first chemical protocol language, although is, to our knowledge, the first specific to peptide chemistry. χDL is one of the most developed languages, so here we focus a comparison on that.

χDL is analogous to a general-purpose programming language in the chemistry space and is able to express protocols from a broad spectrum of chemistry domains rather than just peptides [17]. The same language can execute on multiple diverse hardware architectures, including self-optimising reaction engines [17, 23].

Using XML-style syntax, χDL allows defining a reaction blueprint for a generic synthesis procedure, *e.g.* peptide conjugation, including equivalent amounts for any reagents used. Both specific and generic/placeholder hardware and reagents can be declared. The reaction blueprint can be called multiple times at different scales by passing in the specific reagents to be used, hardware, parameters (duration, temperature), along with the overall reaction scale.

`TideScript` instead focuses on the sub-domain of peptide chemistry, and is optimised for succinct expression of peptide protocols. We take a more declarative approach, where peptide operations produce peptides that can be located physically at a later point. In contrast, χDL's uses a direct treatment of reactors that is reminiscent of C-style explicit memory management.

The domain specific nature of `TideScript` allows it to be more concise when expressing peptide protocols. For example the when expressing a protocol for conjugation, `TideScript` required 13 lines of code (figure 1) versus χDLs 47 (figure 13B). This is mainly due to the XML syntax, additional component declarations, and declaring specific hardware locations; these provide flexibility but are not necessarily required for peptide only chemistry.

As `TideScript` works in a more constrained domain, it can track specific values, *e.g.* since it distinguishes peptides from solvents, properties such as molecular weight, or amino acid sequences can be tracked and inferred. This information could be propagated into a strong future type system to detect poorly formed peptide protocols. One option might be tracking the binding points of specific peptides to ensure a conjugation is possible, *e.g.* there is an unprotected cysteine residue.

Although a formal mapping from `TideScript` to χDL steps has not yet been implemented, this type of mapping should be possible in principle, and could allow χDL to be used as a back-end for `TideScript` to target execution on Chemputer platform.

8.2 BioScript

Digital microfluidic boards (DMFBs) are a technology that utilises electrowetting for liquid handling. Sequential activation of electrodes manipulates fluids, and discrete droplets can be dispensed, mixed, moved and split. Each droplet instance acts as a reservoir for an individual chemical process, and they allow for small scale automation of biological and chemical procedures. DMFBs have already seen application in some automated analytical chemistry and bioanalysis [26].

Many programmable DMF devices have been developed, including OpenDrop [1], Dropbot [12], among others [38]. While there are differences, their application is broadly similar, in that the firmware can interpret inputs (usually in the form of bytes arrays) and execute instructions.

Bioscript is a DSL developed for targeting DMFBs, with an aim to provide a language accessible to life science researchers without programming experience. The compiler employs operation scheduling, module placement and routing of droplets. The scheduler determines the timing of each operation, while considering the resource constraints of the device. Bioscript also introduces a bespoke type-checker that addresses the unique challenges of biochemical operations. The type system cross-references databases from the Environmental Protection Agency (EPA) and the National Oceanic and Atmospheric Administration (NOAA), which categorises 9'800 chemicals into 68 reactivity groups. Many substances will contain multiple functional groups, so a union type is associated with each material. Reactions between certain functional groups can produce other functional groups, so reaction outcomes are calculated in a table, and unsafe reactions can be prevented [26].

8.2.1 Comparing TideScript with BioScript.

BioScript has the most concise expression of the conjugation protocol from reference [30] (see figure 13), compared to TideScript (see reference 1), but unlike χDL and TideScript, it is not hardware agnostic and only targets digital microfluidic boards. Due to nature of the hardware, it can only be used for analytical experiments, and cannot be used to synthesis materials of appreciable quantity [26].

While it can define functions that are compiled to DMF boards, and even supports recursion, they cannot be called at different scales [18].

If a back-end targeting DMFBs was developed for TideScript, then the ability to express protocols in both TideScript and BioScript would be broadly similar. BioScript has a rich type system for preventing mixing of unsafe chemicals. This is for general chemistry, but such a typing system could be of great benefit to TideScript and peptide chemistry as well. A type system like this could improve safety of procedures, particularly if we move towards autonomous systems. TideScript however, has scope for better capturing the peptide-specific domain and its unique challenges.

8.2.2 Modelling of Biochemistry Protocols.

Luca Cardelli *et al* developed a language for modelling and optimising experimental biological protocols. The aim of this research was to pair computational modelling with of the underlying system, with the laboratory protocol execution, in an integrated description. This integration allows models to be formally verified and falsified by taking into account equipment tolerances, and uncertainties in model parameters and data collection [6].

The language incorporated the use of probabilistic semantics, in which uncertainties are characterised by a Gaussian process, using a linear noise approximation. The language primarily models how the respective concentration of species in a chemical reaction network change over time, with temperature and time being parameters to optimise for [6].

Researchers validated their model through simulations of two common biological protocols: *Gibson Assembly*, for joining strands of DNA, and *split and mix* protocols for liquid handling [6]. We discuss the stochasticity of chemistry and how it pertains toTideScript in section 9.2

9 Discussion

TideScript aims to provide a DSL for synthesis protocols focused on the sub-domain of *peptide synthesis*. TideScript's primary research goals are to explore whether targeting a narrower domain in a protocol language unlocks more powerful analyses, and to provide an expressive and accessible

```
manifest peptide
manifest mol_17

step_1 = mix
        peptide with
        mol_17

heat step_1 at 21C for 3h
```

(A)

```
<Synthesis>
     <Hardware>
            <Component
                 id="reactor" type="reactor"/>
    </Hardware>

    <Reagents>
        <Reagent
            name = "SAD(acm)LKSC(acm)LRMCPDRII"
            molecular_weight = "1888.14 g/mol"
            concentration = "0.00106 M"/>

        <Reagent
            name = "molecule_17"
            molecular_weight = "457.5 g/mol"
            concentration = "0.00287 M"/>
    </Reagents>

    <Procedure>
        <Conjugation
            reactor = "reactor"
            peptide = "SAD(acm)LKSC(acm)LRMCPDRI"
            maleimide = "molecule_17"
            time = "3h"
            equiv_amount = "1 mmol"/>

        <Conjugation
                reactor = "reactor"
                peptide ="SAD(acm)LKSC(acm)LRMCPDRI"
                maleimide = "molecule_17"
                time = "3h"
                equiv_amount = "10 mmol"/>

        <Conjugation
                reactor = "reactor"
                peptide = "SAD(acm)LKSC(acm)LRMCPDRI"
                maleimide = "molecule_17"
                time = "3h"
                equiv_amount = "100 mmol"/>
    </Procedure>
</Synthesis>
```

(B)

Figure 13. Comparison of the thiol-maleimide procedure from reference [30] expressed in Bioscript (A) and χDL (B). Full code for the *χDL* script found in appendix 1.

syntax and semantics tailored to domain experts without programming experience. While gaps exist in the range of synthesis protocols that can be automated using TideScript, we have shown strong coverage of protocols in its target domain, particularly SPPS and solution-phase reactions.

9.1 Why Not Just a Library?

We developed TideScript as a stand-alone DSL rather than implementing it as a library within an existing language, *i.e.* an embedded DSL. This choice provides the flexibility to control the syntax and semantics: Deeply embedding peptides as a first-class language construct. Introducing peptides as a specific compiler-aware type enables future analyses, such as determining when two operations are chemically equivalent and performing program rewriting (more difficult with library calls). It also ensures that the language is accessible to chemists with limited programming experience.

Given the differences in hardware, it remains unclear which standard computational elements should be supported: expressions that can be constant folded are likely okay, but it is unclear how to mix computational variables, and chemical variables; how function calls should work on non stack-based architectures; and how existing exception handling paradigms can be mapped to this new domain. There is likely limited reuse of existing compiler optimisation possible. With `TideScript` being relatively simple compared to full programming languages, implementing the toolchain from scratch will likely be less onerous. DSLs have been shown to optimisation for heterogenous CPU/GPU architectures than native libraries [31], and as laboratory hardware is highly heterogenous we believe the improved optimisations[2] to be beneficial here.

An example of a protocol language implemented as a library is Biocoder: a language for modelling biological protocols distributed as a C++ library. The authors' rationale for choosing a library rather than stand-alone DSL was to avoid life scientists learning a completely *new* language. We take the view that end users not proficient in programming languages will still face a barrier using a conventional programming language like C++ [2].

Exploring the implementation details of protocol languages is an interesting area of future research. Some of the principal questions include whether an embedded DSL — whether as a library or taking advantage of meta-programming — is sufficient; or whether the hardware and abstraction differences require developing entirely bespoke language toolchains.

9.2 Chemistry as a Stochastic Process

`TideScript` is symbolic in nature: specifying precisely what steps a protocol should take. It does not contain a model of the underlying chemistry which is a inherently stochastic process: when two chemicals mix the outcome is not fully determined as, for example, small environment changes affect the yield of the produced chemical. The semantics currently do not model the underlying chemistry, hence they do not model the stochasticity. Without a chemical model, `TideScript` (and other protocol languages) are still useful in formally specifying the general structure of protocols and leading the way to automation. Existing, hand-written, protocols are also symbolic, but are of course fundamental to sharing chemical knowledge.

One of the main benefits of the formalised protocols is that it paves the way in future to start asking questions at the intersection of symbolic process models and underlying chemistry. As outlined in Cardelli's work, discussed in section 8.2.2. Such an approach could be integrated within `TideScript`, again the reduction to peptide only chemistry narrowing the amount of chemical modelling needed.

As the outcomes are probabilistic in nature, another interesting question is whether probabilistic programming can capture notions of chemistry in a principled way, *e.g.* using Gaussian processes to model the random variance.

From a runtime perspective, `TideScript` currently does not have a feedback system. For example, we cannot express "stir until molecule is observed". In practice the feedback possible is highly intertwined with the hardware, *e.g.* is NMR available[3].

There could be multiple ways to integrate feedback into `TideScript`: a simple approach is to add conditional statements with the caveat that these might not work on all hardware. It should also be possible to implement feedback as an exception style system, *e.g.* for each mix etc. determine if suitable bounds were met and if not allow the system to respond. A disadvantage of this is the lack of sequential control flow (that chemists are familiar with), but it would allow time savings if earlier parts of a protocol could be automatically reused without starting the entire protocol again.

Additional stochasticity comes from the *conformance* of the equipment (how closely a systems behaviour matches pre-defined models, *e.g.* if we request 20 degrees how close do we actually get).

Probabilistic conformance of cyber-physical systems has been studied, defined by specifications of signal temporal logic (STL), to study powertrain systems, lane-keeping controllers, and power grid systems [36]. There is always a tension between control algorithms and the phenomena they control.

10 Conclusions and Future Work

We have presented the design of `TideScript`, a novel domain-specific language for expressing peptide synthesis protocols. `TideScript` is designed to be accessible to chemists while providing well defined syntax and semantics to allow future strong reasoning. Other than BioScript, a language specific to digital microfluidic platforms, there is limited rigorous programming language design applied to chemical protocol languages; we believe our contribution is a step towards wider application of programming to chemistry protocols.

[2]An interesting question, not discussed here, is specifically what chemical hardware should be optimised for. Reactions are generally long running, and so unlike for computational workloads, time is not a key metric. Instead we could consider optimising for chemical purity, reduced waste, less clean up etc.

[3]NMR is an analytical technique in chemistry that can be used to identify chemical compounds.

We show that TideScript is useful by encoding several realistic peptide protocols from the literature and comparing TideScript with other chemical programming languages: χDL and BioScript.

A core feature of TideScript is to remain hardware-agnostic and instead specify core operations, including movement, mixing, agitating and heating of peptides. Future work will aim to map these operations to specific hardware; for example, agitation could be provided by a magnetic stirrer or a microwave depending on the specific setup. While the basic TideScript is limited to manipulation of peptides in solution, through a small extension—requiring only 4 new constructs and 9 new semantic rules—it can express solid phase peptide synthesis.

While TideScript currently only provides formal specification of protocol operations, for future work we will create a compiler—a prototype version is available [4]—to target chemical hardware. This will start with digital microfluidics, which are suitable for in-solution peptide chemistry, to execute real world experiments. We aim to maintain modularity to allow the possibility to develop back-ends for other hardware like the Chemputer.

An open question is how to take advantage of sensor input at runtime, including developing runtime systems that can respond to analytical data, with the goal of moving towards autonomous cyber-physical systems. This modelling can be paired with models of underlying chemistry, *e.g.* using probabilistic semantics [6].

At the language level, there is untapped potential for exploring the role of type systems to express chemical safety and reliability, as well as avoiding common errors. We hope that deploying this type of analysis more widely can lead to pro-active experimental decisions that can both increase experimental success rates and aid in designing more efficient protocols.

Acknowledgments

This work is supported by The Leverhulme Trust under Grant No. ECF 2021-298, The University of Glasgow Lord Kelvin–Adam Smith Leadership Fellowship, The University of Glasgow Collaborative PhD scheme, and an Amazon Research Award on Automated Reasoning.

References

[1] Mirela Alistar and Urs Gaudenz. 2017. OpenDrop: An Integrated Do-It-Yourself Platform for Personal Use of Biochips. *Bioengineering* 4, 2 (June 2017), 45. doi:10.3390/bioengineering4020045 Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[2] Vaishnavi Ananthanarayanan and William Thies. 2010. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering* 4, 1 (Nov. 2010), 13. doi:10.1186/1754-1611-4-13

[3] Michael N. R. Ashfold, N. Hendrik Nahler, Andrew J. Orr-Ewing, Olivier P. J. Vieuxmaire, Rachel L. Toomes, Theofanis N. Kitsopoulos, Ivan Anton Garcia, Dmitri A. Chestakov, Shiou-Min Wu, and David H. Parker. 2006. Imaging the dynamics of gas phase reactions. *Phys. Chem. Chem. Phys.* 8 (2006), 26–53. Issue 1. doi:10.1039/B509304J

[4] Nicola L. Bell, Florian Boser, Andrius Bubliauskas, Dominic R. Willcox, Victor Sandoval Luna, and Leroy Cronin. 2024. Autonomous execution of highly reactive chemical transformations in the Schlenkputer. *Nature Chemical Engineering* 1, 2 (Feb. 2024), 180–189. doi:10.1038/s44286-023-00024-y Publisher: Nature Publishing Group.

[5] Eris Cani, Tae Eun Park, and Rebecca Kavanagh. 2019. Chapter 27 - Antiviral Drugs. In *A Worldwide Yearly Survey of New Data in Adverse Drug Reactions*, Sidhartha D. Ray (Ed.). Side Effects of Drugs Annual, Vol. 41. Elsevier, 301–319. doi:10.1016/bs.seda.2019.10.005

[6] Luca Cardelli, Marta Kwiatkowska, and Luca Laurenti. 2021. A Language for Modeling and Optimizing Experimental Biological Protocols. *Computation* 9, 10 (Oct. 2021), 107. doi:10.3390/computation9100107 Number: 10 Publisher: Multidisciplinary Digital Publishing Institute.

[7] cdk8s Team. 2025. Escape Hatches - cdk8s. https://cdk8s.io/docs/latest/basics/escape-hatches/

[8] Rosaria Ciriminna, Giuseppe Angellotti, Giovanna Li Petri, and Mario Pagliaro. 2024. Reproducibility in chemistry research. *Heliyon* 10, 14 (July 2024), e33658. doi:10.1016/j.heliyon.2024.e33658

[9] J. Michael Conlon. 2007. Purification of naturally occurring peptides by reversed-phase HPLC. *Nature Protocols* 2, 1 (Jan. 2007), 191–197. doi:10.1038/nprot.2006.437 Publisher: Nature Publishing Group.

[10] Timothy R Deer, Jason E Pope, Michael C Hanes, and Gladstone C McDowell. 2019. Intrathecal Therapy for Chronic Pain: A Review of Morphine and Ziconotide as Firstline Options. *Pain Medicine: The Official Journal of the American Academy of Pain Medicine* 20, 4 (April 2019), 784–798. doi:10.1093/pm/pny132

[11] Aroa Duro-Castano, Inmaculada Conejos-Sánchez, and María J. Vicent. 2014. Peptide-Based Polymer Therapeutics. *Polymers* 6, 2 (Feb. 2014), 515–551. doi:10.3390/polym6020515 Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[12] Ryan Fobel, Christian Fobel, and Aaron R. Wheeler. 2013. DropBot: An open-source digital microfluidic control system with precise control of electrostatic driving force and instantaneous drop velocity measurement. *Applied Physics Letters* 102, 19 (May 2013), 193513. doi:10.1063/1.4807118

[13] Jessica Forbes and Karthik Krishnamurthy. 2025. Biochemistry, Peptide. In *StatPearls*. StatPearls Publishing, Treasure Island (FL).

[14] The Cronin Group. 2025. Full Steps Specification— xdl documentation. https://croningroup.gitlab.io/chemputer/xdl/standard/full_steps_specification.html#evacuateandrefill

[15] Alexander J. S. Hammer, Artem I. Leonov, Nicola L. Bell, and Leroy Cronin. 2021. Chemputation and the Standardization of Chemical Informatics. *JACS Au* 1, 10 (Oct. 2021), 1572–1587. doi:10.1021/jacsau.1c00303 Publisher: American Chemical Society.

[16] Tomas Hardwick and Nisar Ahmed. 2020. Digitising chemical synthesis in automated and robotic flow. *Chemical Science* 11, 44 (Nov. 2020), 11973–11988. doi:10.1039/D0SC04250A Publisher: The Royal Society of Chemistry.

[17] Artem I. Leonov, Alexander J. S. Hammer, Slawomir Lach, S. Hessam M. Mehr, Dario Caramelli, Davide Angelone, Aamir Khan, Steven O'Sullivan, Matthew Craven, Liam Wilbraham, and Leroy Cronin. 2024. An integrated self-optimizing programmable chemical synthesis and reaction engine. *Nature Communications* 15, 1 (Feb. 2024), 1240. doi:10.1038/s41467-024-45444-3 Publisher: Nature Publishing Group.

[18] Tyson Loveless, Jason Ott, and Philip Brisk. 2021. Time- and resource-constrained scheduling for digital microfluidic biochips. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems (ICCPS '21)*. Association for Computing Machinery, New York, NY, USA, 198–208. doi:10.1145/3450267.3450547

[4] https://github.com/NicholasMorris-1/TideScript

[19] Jia-Min Lu, Jian-Zhang Pan, Yi-Ming Mo, and Qun Fang. 2024. Automated Intelligent Platforms for High-Throughput Chemical Synthesis. *Artificial Intelligence Chemistry* 2, 1 (June 2024), 100057. doi:10.1016/j.aichem.2024.100057

[20] Manoj Kumar Mahapatra, Muthukumar Karuppasamy, and Biswa Mohan Sahoo. 2022. Semaglutide, a glucagon like peptide-1 receptor agonist with cardiovascular benefits for management of type 2 diabetes. *Reviews in Endocrine & Metabolic Disorders* 23, 3 (2022), 521–539. doi:10.1007/s11154-021-09699-1

[21] Stephen Kell David Leopoldseder Hanspeter Mössenböck Manuel Rigger, Stefan Marr. [n. d.]. An Analysis of X86-64 Inline Assembly in C Programs. https://dl.acm.org/doi/epdf/10.1145/3186411.3186418. doi:10.1145/3186411.3186418

[22] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. *SIGPLAN Not.* 50, 10 (Oct. 2015), 695–710. doi:10.1145/2858965.2814313

[23] S. Hessam M. Mehr, Matthew Craven, Artem I. Leonov, Graham Keenan, and Leroy Cronin. 2020. A universal system for digitization and automatic execution of the chemical synthesis literature. *Science* 370, 6512 (Oct. 2020), 101–108. doi:10.1126/science.abc2986 Publisher: American Association for the Advancement of Science.

[24] Lisa Nichols. 2017. 1.3C: Transferring Methods - Inert Atmospheric Methods. https://chem.libretexts.org/Bookshelves/Organic_Chemistry/Organic_Chemistry_Lab_Techniques_(Nichols)/01%3A_General_Techniques/1.03%3A_Transferring_Methods/1.3C%3A_Transferring_Methods_-_Inert_Atmospheric_Methods

[25] Yuya Orito. 2025. Accurate and Scalable Prediction of a Fast and Highly Exothermic Nonlinear Reaction System: Reaction Development Using Coupled Simulation of a Mechanism-Oriented Kinetic Model and a Customized Heat Removal Model. *Organic Process Research & Development* 29, 7 (July 2025), 1757–1765. doi:10.1021/acs.oprd.5c00107 Publisher: American Chemical Society.

[26] Jason Ott, Tyson Loveless, Chris Curtis, Mohsen Lesani, and Philip Brisk. 2018. BioScript: programming safe chemistry on laboratories-on-a-chip. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–31. doi:10.1145/3276498

[27] Juan Manuel Parrilla-Gutierrez, Abhishek Sharma, Soichiro Tsuda, Geoffrey J. T. Cooper, Gerardo Aragon-Camarasa, Kevin Donkers, and Leroy Cronin. 2020. A programmable chemical computer with memory and pattern recognition. *Nature Communications* 11, 1 (March 2020), 1442. doi:10.1038/s41467-020-15190-3 Publisher: Nature Publishing Group.

[28] Miguel Quirós, Saulius Gražulis, Saulė Girdzijauskaitė, Andrius Merkys, and Antanas Vaitkus. 2018. Using SMILES strings for the description of chemical connectivity in the Crystallography Open Database. *Journal of Cheminformatics* 10, 1 (May 2018), 23. doi:10.1186/s13321-018-0279-6

[29] Robert Rauschen, Mason Guy, Jason E. Hein, and Leroy Cronin. 2024. Universal chemical programming language for robotic synthesis repeatability. *Nature Synthesis* 3, 4 (April 2024), 488–496. doi:10.1038/s44160-023-00473-6 Publisher: Nature Publishing Group.

[30] James R. Sayer, Karin Walldén, Hans Koss, Helen Allan, Tina Daviter, Paul J. Gane, Gabriel Waksman, and Alethea B. Tabor. 2021. Design, synthesis, and evaluation of peptide-imidazo[1,2-a]pyrazine bioconjugates as potential bivalent inhibitors of the VirB11 ATPase HP0525. *Journal of Peptide Science* 27, 10 (2021), e3353. doi:10.1002/psc.3353 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/psc.3353.

[31] Kai Selgrad, Alexander Lier, Jan Dörntlein, Oliver Reiche, and Marc Stamminger. [n. d.]. A High-Performance Image Processing DSL for Heterogeneous Architectures. ([n. d.]).

[32] Sebastian Steiner, Jakob Wolf, Stefan Glatzel, Anna Andreou, Jarosław M. Granda, Graham Keenan, Trevor Hinkley, Gerardo Aragon-Camarasa, Philip J. Kitson, Davide Angelone, and Leroy Cronin. 2019. Organic synthesis in a modular robotic system driven by a chemical programming language. *Science* 363, 6423 (Jan. 2019), eaav2211. doi:10.1126/science.aav2211 Publisher: American Association for the Advancement of Science.

[33] The Blue Obelisk Team. 2024. OpenSMILES Home Page. http://opensmiles.org/

[34] Sri Murugan Poongkavithai Vadevoo, Smriti Gurung, Hyun-Su Lee, Gowri Rangaswamy Gunassekaran, Seok-Min Lee, Jae-Won Yoon, Yun-Ki Lee, and Byungheon Lee. 2023. Peptides as multifunctional players in cancer therapy. *Experimental & Molecular Medicine* 55, 6 (June 2023), 1099–1109. doi:10.1038/s12276-023-01016-x Publisher: Nature Publishing Group.

[35] Lei Wang, Nanxi Wang, Wenping Zhang, Xurui Cheng, Zhibin Yan, Gang Shao, Xi Wang, Rui Wang, and Caiyun Fu. 2022. Therapeutic peptides: current applications and future directions. *Signal Transduction and Targeted Therapy* 7, 1 (Feb. 2022), 1–27. doi:10.1038/s41392-022-00904-4 Publisher: Nature Publishing Group.

[36] Yu Wang, Mojtaba Zarei, Borzoo Bonakdarpoor, and Miroslav Pajic. 2021. Probabilistic conformance for cyber-physical systems. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems* (Nashville, Tennessee) *(ICCPS '21)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/3450267.3450534

[37] David Weininger. 1988. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences* 28, 1 (Feb. 1988), 31–36. doi:10.1021/ci00057a005 Publisher: American Chemical Society.

[38] Xing Xu, Linfeng Cai, Shanshan Liang, Qiannan Zhang, Shiyan Lin, Mingying Li, Qizheng Yang, Chong Li, Ziyan Han, and Chaoyong Yang. 2023. Digital microfluidics for biological analysis and applications. *Lab on a Chip* 23, 5 (March 2023), 1169–1191. doi:10.1039/D2LC00756H Publisher: The Royal Society of Chemistry.

[39] Mindaugas Šiaučiulis, Christian Knittl-Frank, S. Hessam M. Mehr, Emma Clarke, and Leroy Cronin. 2024. Reaction blueprints and logical control flow for parallelized chiral synthesis in the Chemputer. *Nature Communications* 15, 1 (Nov. 2024), 10261. doi:10.1038/s41467-024-54238-6 Publisher: Nature Publishing Group.

Received 2025-04-25; accepted 2025-08-11