# Work-in-progress: Input Synthesis for Data-dependent Grammars

Simon Winwood
*Galois, Inc.*
Portland, OR, USA
sjw@galois.com

Mike Dodds
*Galois, Inc.*
Portland, OR, USA
miked@galois.com

***Abstract*—We present Talos, a work-in-progress tool for synthesizing examples of data formats encoded in the DaeDaLus data-description language. Talos is based on a multi-layer design: we first analyze a format to identify independent *slices*, and we then use search and SMT solving to synthesize solutions for these slices. Talos works for small examples of DaeDaLus formats and we are working on scaling it to two target examples: the PDF and NITF format descriptions we have previously encoded in DaeDaLus.**

## I. Introduction

The challenge of constructing a correct parser for a format may be (very roughly) broken into two problems:

1) Specifying the format correctly, i.e. the one that is desired by the users of the system. This is typically achieved using some kind of format definition language.
2) Constructing a parser from this specifications in a correct, secure, and performant way. This is typically achieved using some mix of hand-written code and automated parser generation.

Existing systems struggle with both of these objectives when presented with real-world formats that feature data dependency. To help solve both of these problems, Galois is developing DaeDaLus[1], a system for constructing parsers for complex data-dependent formats. DaeDaLus takes the form of a high-level format definition language and a suite of tools for generating parsers from format definitions (by convention, we refer to both the language and toolchain as DaeDaLus).

DaeDaLus helps solve problem (2) by providing a strongly typed input language which precludes many kinds of errors, and a high-assurance compiler based on the memory-safe language Haskell. DaeDaLus also helps solve problem (1) by providing a precise and compact format for defining parsers. This means that parsers can be audited more easily by domain experts. Testing is another mechanism for addressing problem (1). DaeDaLus supports testing of inputs which are generated by some other process, which DaeDaLus either parses or rejects. For example, this allows DaeDaLus to check conformance against an existing test suite, or the output of a generator.

However, we envisage another mode of parser testing which would also help solve problem (1). In this mode, sample documents are generated from the DaeDaLus format description itself. Ideally, documents that are generated in this mode should achieve a high coverage of the format in order to exercise all corner cases. The documents that are generated can then either be inspected by hand, or used to test other parsers that target the same format. If discrepancies are discovered, the original format description can be amended to better reflect the desired format. This is the problem we are attempting to solve with our new tool, Talos[2].

In this paper we present our design for Talos, and some early results. As well as its value as an independent tool, Talos also demonstrates the usefulness of the DaeDaLus approach to format definition. Data-dependent properties are captured cleanly in DaeDaLus, rather than being expressed through ad-hoc code. As a result, it is possible to build supporting tools like Talos which can reason about data-dependency as a first-class property of the format.

### A. Talos Use Cases

Talos is a generic framework for synthesizing examples of formats encoded in the DaeDaLus language. As such, we envisage several possible use-cases for Talos in document definition processes.

- *Testing a DaeDaLus format description*. In this scenario, the user writes a DaeDaLus format description which attempts to capture a format that has other existing implementations. Talos can then be used to test the DaeDaLus definition by generating test cases, which can then be parsed by the existing implementations.
- *Testing a hand-written parser*. In this scenario, the user hand-writes a new parser in (for example) C++, and in parallel writes a DaeDaLus specification which defines the desired behavior of the parser. Talos can be used generate test-cases that the C++ parser conforms to the specification.

This work was funded by the DARPA SafeDocs project, DARPA Award HR001119C0073.

[1]https://github.com/GaloisInc/daedalus/

[2]We note that the current version of Talos only supports the generation of positive examples, i.e. examples which are inside the format as currently defined. An equally important but more difficult problem is the generation of *negative* examples, which we discuss briefly in subsection V-C at the end of the paper.

- *Testing a parser's context*. A parser is not a stand-alone application, and so any parser is typically surrounded by an application which consumes the results of the parser. In this scenario, Talos can be used to generate documents which, after parsing, can be used to test the surrounding application.

The common thread in these applications is that Talos can help ensure that the DaeDaLus that has been written is correct. By allowing DaeDaLus to be a parser and a generator, we can assist programmers in getting their specifications correct. However, building Talos presents several major challenges. Because DaeDaLus is a fully featured programming language, generating input documents resembles the problem of generating valid program inputs, a problem known to computationally very difficult.

### B. Talos overview

A key feature of DaeDaLus which distinguishes it from traditional parser generators is that one can encode *data dependent languages*, where a grammar can examine the semantic values constructed by other parts of the grammar. This is useful, for example, for describing tables of values preceded by the table length. DaeDaLus supports a rich semantic language, including bitvectors, sums and products, along with recursive types. Furthermore, it is straightforward in DaeDaLus to write predicates over these types; a specification can demand that a list of values is sorted, for example.

Data-dependence increases the descriptive power of DaeDaLus, at the cost of greatly complicating document synthesis. Unlike a context free grammar where a document can be generated by simply choosing productions, in DaeDaLus a poorly chosen value can make synthesizing the remainder of the document impossible. The main problem we are addressing is then how to choose values that satisfy user-specified, and hence arbitrary, relations.

In general, synthesis for DaeDaLus is undecidable as the DaeDaLus language is Turing-complete. However, we hypothesize that the formats we are interested in supporting in DaeDaLus are amenable to synthesis. The primary research problem for Talos is then how to perform synthesis *efficiently*.

The reason we hypothesize that synthesis in Talos is feasible is due to a key insight: even where data-dependency is present in a format, the majority of the format is usually non-data-dependent. For example, the length of an array may be dependent, but each separate cell of the array is likely to be non-dependent. Furthermore, the data-dependency used in these formats is not global, in that length fields, for example, can be chosen independently of other dependent values. Finally, the types of computation in these formats tends to be simple, such as concatenating bitvectors, or checking values are in (numerical) ranges. Thus, synthesis for a data-format is in general equivalent to general program input synthesis, but in practice it is much easier for the real-world formats that we have examined.

At the core of Talos is a constraint-solving problem: a set of bytes must be synthesized that are mutually consistent according to the format specification. Talos admits several strategies for doing this, for example, backtracking search and symbolic execution / SMT solving [1]. However, complete formats are typically too large to address in a single step. To solve this, Talos uses a multi-layered strategy. It first performs an initial dependency analysis which identifies the naturally independent portions of the format; these so-called *slices* [2] can be synthesized separately. This is what allows us to synthesize real-world formats in a performant manner.

## II. Related Work

The motivation and approach take by Talos is similar to a number of other techniques. Perhaps the closest example is SMT (and SAT) solving, where the solver seeks to construct a model from a problem expressed in logic. We experimented with embedding the synthesis problem as a SMT problem, including recursive definitions and inductive data structures. This approach showed promise for smaller problems, but did not scale. In particular, while recursive definitions and types are supported by SMT-LIB [3], and implemented in solvers such as Z3 [4], in practice these features are experimental and unpredictable. For example, Daedalus's Many combinator, which executes a parser multiple times, worked well for simple parsers, but failed to terminate when nested.

Talos's approach to generating inputs is similar to the *dynamic symbolic execution* [5] approach used by systems like DART [6] and KLEE [7]. This approach symbolically execute the target application to generate concrete inputs which, for example, lead to an assertion violation. While KLEE is primarily focused on white-box testing of imperative programs, the search techniques used should be applicable to Talos. An interesting difference in the application domains between KLEE and Talos is that the input problems to Talos tend to be much smaller and simpler to reason about, due to the high-level and purely functional nature of the DaeDaLus language. Thus, Talos can perform a pre-processing analysis step to inform the symbolic execution stage, whereas this may be prohibitive for KLEE-style systems.

Many other systems exist, such as miniKanren [8] that are designed to synthesize solutions to constraint sets of various kinds. It is natural to wonder whether Talos could be implemented as a light encoding to one of these systems. In fact, this was our original approach to Talos, but it proved to be non-scalable. The key challenge with Talos lies in optimization based on the structure of the format. A naive encoding of a format into (for example) miniKanren rapidly becomes unscalable. Instead, what proved to work was Talos's current two-layer design: an analysis that identifies natural independencies in the format, and a search process which is backed by SMT solving. In fact, other constraint solving approaches could be plugged in as well as SMT - we plan to experiment with these in future

The complexity of synthesis in Talos is created by data dependency. A non-dependent format can be encoded as a context-free grammar. In this case, synthesis is simple [9],

[10]: we just select the productions of the grammar. Fortunately, even in dependent formats, it is often true that portions of the format are non-data-dependent. In this case, the simple strategy applies: discovering where this is true is one of the roles of the analysis performed by Talos. However, in many key areas of a format, dependency plays a critical role in defining what documents are valid. In this case, we need to use solver-backed search to generate solutions

## III. Talos in more detail

Talos operates over format descriptions written in DaeDaLus's input language. This is a fully featured functional programming language. However in this section we ignore most of these features and present examples in a heavily simplified form. We write our examples in a very small language we call micro-DaeDaLus. Figure 1 shows the grammar for micro-DaeDaLus.

We leave the semantics of this language unformalized, but we hope it is reasonably intuitive to those familiar with functional programming languages. Our examples also include `let`-binding and a `Block` construct sequencing multiple operations. These are encoded in the obvious way.

### A. Data-dependency in Talos

Here is a micro-DaeDaLus definition for a format consisting of two ordered bytes.

```
def Example1 =
  block
    let x = Byte
    let y = Byte
    Guard x < y
```

This format binds two variables to bytes read from the bytestream, and then applies a constraint on the bytes that the first is numerically less than the second. This format will recognize many two-byte streams: for example `0x00,0x01`, or `0x23,0x42`.

Here is another small DaeDaLus format which demonstrates the use of the non-deterministic choice operator:

```
def Example2 =
  block
    let x = Choice
              Pure 0xff
              Byte
    let y = Byte
    Guard x < y
```

In this example, the first assignment chooses between the constant value `0xff` and reading from the input stream. This example illustrates that DaeDaLus formats are inherently non-deterministic, and that some of the paths through the format may not be viable. The runtime cannot pick `0xff` as the value of `x`, and it will backtrack if it makes this particular choice. This format recognizes the same set of two-byte streams as the previous one.

### B. Synthesis

A very naive version of Talos would just exhaustively enumerate possible bytestreams until one is discovered that can be recognized. This approach is sound, complete, but unacceptably slow for real formats. However, it illustrates that building Talos is fundamentally an *optimization exercise*: the majority of the effort in developing Talos is involved in dividing the problem into simpler sub-problems, and optimizing the synthesis strategies.

Talos supports multiple strategies for synthesizing bytestreams. The simplest of these is just depth-first search: the values are selected at random in the order they appear in the format, with backtracking occurring on a guard failure. This strategy performs well when the solution space is dense, such that few guesses are required.

A more sophisticated strategy is to encode the guard into an SMT query, which is then handed off to a solver. The current implementation of Talos uses random depth-first search for selecting paths through the format, unfolding recursive calls as required, and sends a query to the solver whenever a guard is encountered. An alternate approach is to symbolically execute the entire format, using the solvers support for recursive functions to model recursion in the input format. This approach requires experimental solver features (such as recursion) which, in practice, do not work well for the sorts of problems that Talos poses.

### C. Analysis

Whole-program format synthesis does not scale beyond trivial examples. The DaeDaLus language is as expressive as a general language such as Haskell, which means that we can easily construct pathological examples where every program variable interacts with every other variable. However, our experience with real formats is that they are composed of loosely coupled pieces that are not mutually dependent.

Our aim with Talos is to discover structure within the format that allows us to synthesize bytestreams more quickly. Thus, before synthesis, Talos constructs a collection of simpler formats termed *slices*. A slice is essentially an erasure of the input format, where uninteresting parts of the format are replace by holes.

Each slice corresponds to a collection of *entangled* variables, where two variables are entangled if the choice of one impacts the allowed choices for another. Typically variables which are free in a guard are entangled, although when two variables are entangled is somewhat subtle: consider a case statement, eliminating a variable of sum type, where some alternatives constrain other variables. In this case the discriminated variable is entangled with those from the alternatives.

Consider `Example1`. The values selected for `x` and `y` are mutually dependent: if we select a value for one, this affects the available choices for the other, and so these two variables are entangled. The slices constructed by the analysis correspond to the sets of entangled variables.

To illustrate this idea, consider the following DaeDaLus format:

| | | | |
|---|---|---|---|
| Grammar | ::= | `Byte` | *read a byte from the input stream* |
| | \| | `Pure` Expr | *pure expression* |
| | \| | `Guard` Expr | *conditionally continue executing* |
| | \| | `Choice` Grammar Grammar' | *non-deterministic choice* |
| | \| | `Bind` VName Grammar Grammar' | *variable bind and sequencing* |
| | \| | `Call` FName VName | *function call* |

Fig. 1. Syntax of micro-DaeDaLus.

```
def Example3 =
  block
    let x = Byte
    let y = Byte
    let z = Byte
    Guard x > z
    Guard y > 0x01
```

The variables `x` and `z` are entangled through the guard on line 6, while the variable `y` only affects the guard on line 7. Talos will therefore project two slices from the block:

*Slice 1:*

```
block
  let x = Byte
  [...........]
  let z = Byte
  Guard x > z
  [...........]
```

*Slice 2:*

```
block
  [...........]
  let y = Byte
  [...........]
  [...........]
  Guard y > 0x01
```

It should be clear from this example that bytestreams for each slice can be synthesized separately, and then the answers reconstructed into a single bytestream. A key consequence of the slicing process is that slices are *independent*, and so their corresponding bytestreams can be synthesized in isolation.

This example also illustrates that slices need not be contiguous in the format: two slices may correspond to interleaved code, and so the results of synthesis for a slice need not be contiguous in the resulting bytestream. Furthermore, those parts of the format which do not appear in a slice are, by construction, non-data-dependent, and so the input format can be considered the union of the (disjoint) slices plus the non-dependent remainder.

## D. Synthesis approach

The synthesis algorithm in Talos consists of a two-layer process: an outer function which traverses the syntax of the format, randomly making choices at non-dependent branch-points and values. When the start of a slice is reached, Talos invokes the strategies to find the bytestream fragments which correspond to the slice. We note that each slice in the format is associated with a variable binding: by definition, only named values can be used in a data-dependent fashion, and there is an earliest variable which dominates the remainder of the slice.

Synthesis over slices generates a set of constraints called a *path* which determines both control-flow choices and the value of bytestream reads. As Talos executes the outer function, it makes sure to make corresponding choices to those determined in the path. When it reaches a read from the bytestream, it may be that the value is unconstrained: in this case, Talos chooses a value randomly. Otherwise, the read has been fixed by slice synthesis: in this case, Talos chooses the correct value.

A key property of the analysis and synthesis is that control-flow choices and reads from the bytestream are either predetermined or cannot affect the success of execution. In other words, all dependent nondeterminism is resolved through the inner slice synthesis function, not through the outer execution loop. This means that expensive constraint solving has been localized to just the dependent slices.

## E. More advanced slicing

Example 3 above consists of straight-line code where the two slices are obviously independent. Real Daedalus formats have two features that make construction of slices more complex: branching control flow and function calls.

*a) Branching control flow:* This must be handled carefully because slices are intended to be synthesized independently. In other words, once decisions are made for a particular slice, they should not conflict with decisions made for other slices. This is problematic when two slices cover the same control flow, albeit for different variables. We must not make a choice in one slice which forces the program down an infeasible path for the other slice.

Talos solves this problem conservatively by joining slices when they overlap in control flow. This has the effect of entangling variables that would otherwise be independent, but this cannot be avoided in general, as the choice of input byte can affect another byte through control flow, as well as through guards.

*b) Function calls:* These can be inlined in most cases, in which case the normal slice construction process applies. However, we also use function calls to represent unbounded computations in Daedalus (i.e. instead of explicit loops). In this case, Talos constructs a slice specification for each function which breaks the function down into slices. These slices are used during synthesis to give the correct set of choices for each synthesis calculation.

The slices constructed for a function may be entangled with the function's arguments: this is the analog of entangled variables that are in scope if the function was inlined. Variables inside the function may also be entangled with its calling context through its return value. To solve these issues, Talos constructs three types of slice specification: slices that are purely internal to the function, slices that are entangled with the variable arguments, and a special set of return slices that can generate entanglements through the return value.

As functions may be recursive, Talos performs a fixed-point calculation to discover a consistent slice specification for the function. This fixed point calculation is guaranteed to terminate as slice specifications form a lattice structure.

## IV. Preliminary results

We have successfully applied Talos to simple formats. including the PPM image format[3]. Talos can synthesis a PPM image in typically, under a second. However, Talos can fail to finish in a reasonable time due to (randomly generated) large image sizes. This suggests that tracking data dependencies which impact the amount of work that Talos must do could assist in producing results more predictably: in the case of a table length, for example, choosing large values will require a corresponding large amount of work in producing the table entries.

We are exploring applying Talos to larger formats, including the NITF image format[4], and PDF. In the case of NITF Talos can synthesis the document header, but fails to find a bytestream for the whole format. Several of the extant optimizations, such as the user-supplied inverses, and partial concrete execution during synthesis, were motivated by this example. While we were able to achieve a 5x improvement in synthesis time, we are investigating improvements to the analysis algorithm to produce smaller, more tractable slices to allow Talos to succeed on the whole document.

The PDF specification in DaeDaLus has multiple layers including a specification of the basic objects in the specification, such as strings, numbers, arrays, and dictionaries, and higher level concepts, such as pages and fonts, expressed as predicates over PDF objects. We do not expect Talos to synthesize an entire PDF, at least, not without some assistance. In this scenario we envisage Talos being used as a library which can be used to, for example, produce well-formed replacement objects in a larger PDF. We expect that layers in a format will feature some degree of encapsulation between layers—at least, this is true for PDF. An intriguing question for future work is whether Talos could exploit this encapsulation via some kind of modular analysis.

Multi-level specifications, such as PDF, where the types of values are described generically and then later constrained can lead to clearer, and thus preferable, specifications, but expose weaknesses in the way that Talos generates bytestreams. In particular, the symbolic strategy can be thought of as eagerly selecting control paths (choice in the grammar) but lazily selecting the data values (e.g the bytes). Then, for example, in order to generate a particular PDF object such as a font description, the symbolic execution strategy will first construct some PDF object, and then perform the guard to check that the object has the correct form. This check is unlikely to succeed, and so the synthesis strategy will backtrack and try again.

[3]https://en.wikipedia.org/wiki/Netpbm

[4]https://en.wikipedia.org/wiki/National_Imagery_Transmission_Format

We are in the process of extending the analysis phase to propagate back information about allowed value shapes, that is, the allowed variants for a particular value, such that choices which will never lead to a successful value are ignored. Ideally, Talos will produce the equivalent of a single-level specification for a given PDF object type. In practice, it may be sufficient to propagate an abstraction of the constraints, such as converting a relation into (weaker) predicates.

## V. Future directions

There are a number of intriguing future directions to improve the applicability of Talos. These enhancements can be categorized as allowing Talos to succeed on more formats; to generate more documents faster; and to generate better quality documents.

### A. Successful generation

Strategies typically perform better when given smaller problems: the simplest strategy, random depth-first search, works very well when the probability of a given value being suitable is high. Better analysis can propagate back information allowing the strategies to make better choices: an interval analysis, for example, would effectively remove unsuitable values from the search space, and hence increase the probability of a successful choice.

The search space for many formats is infinite, as recursive values can be unbounded. Thus, depth-first search can get stuck in a branch of the search space which contains no solutions. Alternative search strategies, such as breadth-first search may avoid these issues. These strategies, however, can be rather inefficient as, for example, BFS retains the state for the entire search frontier.

Selecting from a range of search strategies then becomes important: running the strategies in parallel, or sequentially with time outs, are possible solutions, as are learning-based approaches where information is retained about which strategies work for particular slices.

### B. Faster generation

As noted previously, document synthesis is essentially an optimization problem. While the time to synthesize a single document is important, we are also interested in bulk generation. In some usage scenarios, such as testing for equivalence, it is beneficial to have a large number of synthesized documents.

*a) Automatic inverse construction:* Some grammars, such as those defining the format for numbers, result in values which can be directly converted into their bytestreams, but are difficult to construct bottom-up. In the case of constrained numbers, for example, it is simple to pick a number in a range and then produce the digits which encode that number, compared to picking bytes which combine to form a number in the range. Talos currently supports user-supplied grammar inverses in the form of DaeDaLus functions, however automated construction of these inverses should be feasible at least for a subset of DaeDaLus expressions.

*b) Parallel strategies:* Recall that slices are independent, and as such can be generated in parallel, potentially across a large cluster of servers. While extending generating bytestreams for slices in parallel is trivial, selecting which slices to generate for is less so: not all slices occur on all program paths, and so the outer loop would need to be update to first select among the non-dependent choices, and then merge the solutions back into the slices occurring with those choices.

*c) Offline slice solving:* Given an outer-loop which can first select *which* slices need to be solved, without requiring their solutions, Talos could generate a pool of bytestreams for each slice. These could then be combined combinatorially to generate a large number of documents.

### C. Better quality

The development of Talos has focused primarily on the previous categories. However the quality of generated documents is also of interest. For testing, specification coverage is important, as is generating documents distributed uniformly; producing documents containing a mostly zero values, for example, is unlikely to be useful.

For implementation testing, it would be useful for a user to control which parts of the specification are being explored when doing bulk generation of documents, especially where a divergence is discovered. Furthermore, given an existing corpus of test documents, it would be useful if Talos could generate both similar documents, by, for example, mirroring the distribution of choices present in the corpus, and generating documents which are complementary to the existing corpus.

Finally, Talos currently only constructs positive examples, i.e. those that are inside the current format. An equally useful capability would be to generate negative examples, i.e. those that are outside the format. This is inherently a harder problem to specify than positive generation: for example, a random sequence of bytes is likely to be outside the format, but is not useful as a testcase. Instead, we need to synthesize examples that are 'nearly' inside the format. Several ideas may be useful here. We could generate positive examples and then mutate them to try to push them outside the format. Or we could mutate the format itself, and generate positive examples from this new version. It may even be possible to use failures during generation as negative examples. However, these ideas depend on Talos functioning at scale as a source of positive examples, which is our current research focus.

## VI. Conclusions

Talos is a work in progress: we have built a tool in prototype that can solve synthesis problems for small and constrained DaeDaLus programs, and we are working on improved scalability and improved coverage guarantees. As we discuss above, building Talos is fundamentally an *optimization problem*. There is no absolute success criterion, because we will always be able to find pathological parsers which foil the generator. Instead, what we aim for is a tool that performs well in the situations that naturally occur when using DaeDaLus to describe extant data formats such as PDF and NITF.

We note once more that the existence of DaeDaLus as a domain-specific specification language was essential to making Talos feasible. Talos takes advantage of DaeDaLus in multiple ways. It leverages the infrastructure developed as part of the DaeDaLus parser generation effort, including complication phases to simplify the format's AST, and the interpreter to perform concrete evaluation during synthesis. The analysis leverages the side-effect free functional nature of DaeDaLus terms to make generating solver terms feasible. We might compare the problem of synthesis for DaeDaLus to a parser written in a more traditional style, with generated portions and embedded C or Haskell semantic actions. Generating input for these parsers would be infeasibly complicated, whereas for DaeDaLus it is merely a difficult problem that can be solved.

## References

[1] J. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, 07 1976.

[2] M. Weiser, "Program slicing." *IEEE Trans. Software Eng.*, vol. 10, pp. 352–357, 07 1984.

[3] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[4] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[5] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, p. 82–90, feb 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408795

[6] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[8] D. P. Friedman, W. E. Byrd, O. Kiselyov, and J. Hemann, *The Reasoned Schemer*, 2nd ed. The MIT Press, 2018.

[9] P. Purdom, "A sentence generator for testing parsers," *BIT*, vol. 12, pp. 366–375, 01 1972.

[10] A. Celentano, S. Crespi Reghizzi, P. Vigna, C. Ghezzi, G. Granata, and F. Savoretti, "Compiler testing using a sentence generator." *Software: Practice and Experience*, vol. 10, pp. 897 – 918, 11 1980.