Supplemental: Leveraging Language to Learn Program Search Heuristics and Abstractions

This contains the supplemental appendix to the 2021 ICML paper. It is organized sequentially in reference to the main text; S{N} refers back to section N in the main text.
A complete release of code for our implementation, including command line scripts to replicate the experiments in the paper and links to the datasets, can be found at: `https://bit.ly/3g9361W`.

S4. Base learning algorithm: DreamCoder

The LAPS framework described in the main paper (Sec. 5) is a general one for extending Bayesian models of program learning to incorporate information from natural language (see (Liang et al., 2010; Lake et al., 2015; Dechter et al., 2013; Lake et al., 2013)). Our concrete implementation and experiments use the DreamCoder approach of (Ellis et al., 2021; 2018) as the base synthesis algorithm, which implements the hierarchical Bayesian formulation of program learning. It defines a modular interface with two primary learning components: a learned *conditional inference* model for search (as a neural search heuristic); and a learned *abstraction* algorithm for updating the program prior (based on program refactoring and compression) (Ellis et al., 2021). Each of these learning components has been additionally implemented in other work (such as (Devlin et al., 2017; Polosukhin & Skidanov, 2018; Nye et al., 2019; Parisotto et al., 2016; Balog et al., 2016) for neurally guided synthesis, and (Dechter et al., 2013; Zhang et al., 2017; Shin et al., 2019; Artzi et al., 2014; Dumancić & Cropper) for program abstraction learning).

This supplementary section provides theoretical and implementation details on the DreamCoder algorithm we use in our experiments (summarized in Sec. 4). We match our implementation as closely as possible to the original work for comparison with published baselines. We provide key details relevant to the language-guided extension, but strongly recommend the original works which introduce the DreamCoder algorithm (Ellis et al., 2021; 2018) for further reference.

S4.1 Program prior and MDL equivalence

Hierarchical Bayesian program learning formulations require a prior over expressible programs. DreamCoder is learned iteratively: it is initialized with a base library $\mathcal{L}_0$ and returns a library $\mathcal{L}_f$ containing program abstractions learned from solving training tasks. Therefore, DreamCoder defines its program prior with respect to the current library $\mathcal{L}_i$ maintained at each iteration. This is parameterized as a simple PCFG $\mathrm{P}[\rho|\mathcal{L},\theta_{\mathcal{L}}]$ whose productions are of the form $l_i \to l_j \in \mathcal{L}$, each with a real-valued weight $\theta_{\mathcal{L}l}$, where the probability of a program ρ is given by $\mathrm{P}[\rho|\mathcal{L},\theta_{\mathcal{L}}] = \prod_{l\in\rho}\mathrm{P}[l|\mathcal{L},\theta_{\mathcal{L}}]$ (Sec. 4.1).

Minor complexity arises in order to support typing (Pierce, 2002): following (Ellis et al., 2018), the library $\mathcal{L}_i$ is implemented as a set of polymorphically typed λ-calculus expressions. The only change this produces to the original prior definition is to restrict the set of possible productions under the PCFG: that is, permissible productions are of the form $l_i \to l_j \in \{\mathcal{L}|l_i \to l_j \text{is well typed}\}$. The prior probabilities of programs are therefore calculated with respect to the set of well-typed productions.

As discussed in the main paper, this prior definition is *equivalent to a minimum description-length prior over programs* under $(\mathcal{L},\theta_{\mathcal{L}})$ when all $\theta_{\mathcal{L}} < 1.0$, as the product of additional productions in an expression will strictly decrease as the number of productions in an expression increases.

S4.2 Amortized conditional inference

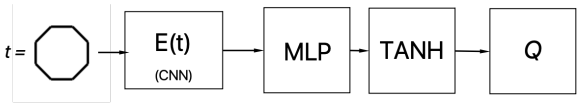

Figure 1. Architecture of the neural model $Q_i(\rho|t,\mathcal{L}_i)$. The model takes as input task examples t. These are encoded using a domain-specific encoder $E(t)$. Task encodings feed to an MLP and activation layer and output a tensor Q. This parameterizes a distribution over program bigrams in the final DSL, which defines a conditional distribution from which to enumerate programs during search.

To identify programs that solve tasks t while obtaining high probability under $\mathrm{P}[\rho|\mathcal{L},\theta_{\mathcal{L}}]$, DreamCoder trains a neural search heuristic $Q_i(\rho|t,\mathcal{L}_i)$ at each iteration i to approximate the inverse model.

The training procedure in (Ellis et al., 2021) (summarized in Sec. 4.2) is a key contribution of the original work for learning in the distant supervision setting. The model is trained on samples from the generative prior (providing an endless

training stream of random synthesis tasks); and this procedure should generalize immediately to any neural model for predicting programs conditioned on the task specification (e.g. (Devlin et al., 2017; Polosukhin & Skidanov, 2018; Nye et al., 2019; Parisotto et al., 2016; Balog et al., 2016)). The model is also supervised on any original training task examples and their program solutions discovered during learning.

In our experiments we use the baseline neural model architecture in (Ellis et al., 2021). This is parameterized by two modular components:

1. *A domain-specific task encoder* $E(t)$. This encodes the task examples (e.g. *images* in the graphics program domain, or input-output strings in the text editing domain) that are input to the neural model. This task encoder architecture is defined domain-specifically based on the form of the task examples (e.g. a CNN for the graphics domain). It outputs a fixed dimensional embedding for any given task as *input* to the model. In our experiments this is a 64-dimensional embedding across all domains (See S6.1 for domain-specific architectures; and released code.)

2. *A conditional model over programs* $Q(\rho|E(t))$. This component receives the task encoding as input and outputs a distribution over programs. Following (Ellis et al., 2021), this is a 2-layer fully-connected MLP (with 64 hidden units and a final tanh activation layer) that outputs a fixed-dimensional real-valued tensor encoding a distribution over programs in the library $\mathcal{L}$ as output. The real-valued tensor corresponds to weights over program primitives conditioned on their local context in the syntax tree of the program, consisting of the parent node in the syntax tree and which argument is being generated. This functions as a 'bigram transition model' over trees that encodes the likelihood of transitions from one primitive to the next. Q returns this as a $(|\mathcal{L}|+1)\times(|\mathcal{L}|+2)\times A$-dimensional tensor, where A is the maximum arity of any primitive in the library.

This parameterization supports fast sampling of programs during conditional synthesis: the neural model runs once per task (to encode the task examples and produce the bigram transition model) and the resulting parameterization can then be used to sample programs during synthesis (e.g. by enumerating programs by expanding trees (as 'bigrams' over parent and children primitives) ranked in order of their likelihood starting from the program root.)

Following (Ellis et al., 2021), the neural model is trained to optimize the following MAP inference objective on the training tasks and the sampled tasks from the prior:

$$\mathcal{L}_{\text{MAP}=\mathrm{E}_{t\sim(\mathcal{L},\theta_{\mathcal{L}})}}\left[\log Q\left(\arg\max_{\rho}\mathrm{P}[\rho|t,\mathcal{L},\theta_{\mathcal{L}}]\;\middle|\;t\right)\right] \quad (1)$$

S4.3 Abstraction learning as program compression

DreamCoder learns new abstractions to approximately optimize for Eq. 2 (main paper), which infers an optimal library and parameters with respect to the observed programs on the training tasks.

The DreamCoder abstraction algorithm is a primary contribution of the original work in (Ellis et al., 2021), and is discussed extensively in (Ellis et al., 2021). We therefore provide additional technical details here that are relevant to its integration with LAPS in our experiments, but strongly encourage referencing (Ellis et al., 2021) for the full implementation.

As discussed in (Ellis et al., 2021) and our main work, DreamCoder approaches abstraction using an equivalence between Eq. 3 and the *minimum description length* of the *prior* (as the description length of the library) and the *programs* produced from the prior (under the PCFG definition of the prior). Therefore, in practice, inferring the optimal library is equivalent to inferring the library which maximally compresses the description length of the library and the description length of programs which explain the training tasks. In particular, DreamCoder optimizes the following compression objective with respect to the training tasks T and the finite *beam* B_t of program solutions discovered for each training task during learning:

$$\log\mathrm{P}[\mathcal{L}]+\arg\max_{\theta_{\mathcal{L}}}\sum_{t\in T}\log\sum_{\rho\in\mathcal{B}_t}\mathrm{P}[t|\rho]\max_{\rho'\rho}\mathrm{P}[\rho'|\mathcal{L},\theta_{\mathcal{L}}] + \log\mathrm{P}[\theta_{\mathcal{L}}|\mathcal{L}]-|\theta_{\mathcal{L}}|_0 \quad (2)$$

The key aspect of this algorithm is that it considers abstractions which compress not only the programs as they are *currently written*, but any semantically equivalent *refactorings* of these programs. Specifically, as programs are written in a λ-calculus, *refactoring* refers to any program which is equivalent up to β-reduction (i.e., function application/variable substitution (Pierce, 2002)). A primary contribution of the original work in (Ellis et al., 2021) is an efficient algorithm for computing these refactorings that is unchanged when we integrate language; we refer to the original text for details.

In our work, the primary important aspect of this aspect is that refactorings are defined compositionally over the existing program primitives. Specifically, refactorings can be efficiently calculated according to semantic equivalences in the the λ-calculus (namely, that function application and variable substitution guarantee that the resulting refactored programs are equivalent. *Abstractions* created by variable

substitution will always be composed of subcomponents from the initial library.) We take advantage of this compositionality when defining our joint abstraction algorithm over natural language. Defining an initial *compositional* translation model between language and the program components ensures that we can approximate compression in the joint model after the programs are refactored, without needing to induce an entirely new translation model over language and the refactored programs.

S5. Our Approach: Language for Abstraction and Program Search

This section now describes technical details for the concrete LAPS implementation in our reported experiments, which is defined over the DreamCoder implementation. We structure this section according to the parallel implementations in the base algorithm for clarity. However, except for the specifics of the joint-abstraction algorithm, the technical implementation of each component should extend directly to most other similar learned synthesis algorithms (e.g. the joint model implementation should be reusable in *any* synthesis algorithm that uses an explicit symbolic library of primitives.)

S5.1 Joint prior over programs and language

LAPS extends the prior $\mathrm{P}[\rho]$ over programs under the library to a *joint* prior $J(\rho, d_t)$ over programs for a given task and their natural language descriptions d_t (Sec. 5.1). We formulate this prior as

$$J(\rho, d_t) = \mathrm{P}[\rho|\mathcal{L}, \theta_{\mathcal{L}}]\mathrm{P}[d_t|\rho, \mathcal{L}]$$

the product of the original prior over programs $P[\rho|\mathcal{L}, \theta_{\mathcal{L}}]$ defined on the program library, and a *program to descriptions* "translation" model $\mathcal{T}(d_t|\rho, \mathcal{L}) \approx \mathrm{P}[d_t|\rho, \mathcal{L}]$ that describes how descriptions are generated for programs written in the library.

The concrete implementation described in the main paper uses a translation model that additionally decomposes compositionally over language and programs–in particular, on the basis of token-token translation distributions $\mathrm{P}_{\mathcal{T}}[w|l]$ between words $w \in d_t$ and $l \in \mathcal{L}$. Many available translation and semantic parsing models (such as synchronous grammars over natural language and programs) preserve this further compositional requirement (e.g. (Artzi et al., 2014; Wong & Mooney, 2006)).

See Figure S3 (supplement) for example samples from the generative model on the graphics domain at earlier and later stages of training.

Our implementation uses a classical statistical machine translation model (the Model 4 version of the IBM Statistical Machine Translation models (Gal & Blunsom, 2013)) whose parameters can be tractably estimated from very few paired programs and descriptions (in the distant supervision setting used in the original work, there may be no more than a couple of hundred training tasks in the full dataset, and fewer than 10 solved tasks on which to train the translation model at any given time.) In addition to inference in small data settings, this translation model has a fully compositional generative definition (Gal & Blunsom, 2013) that allows it to be easily used to train the neural amortized inference model which conditions on language.

Despite this, however, this translation model (and the further inductive biases used to specifically relate program trees to sentences) make strong compositonality assumptions about the relationship between program primitives and words as a joint generative model of programs and language; we find that these inductive biases are useful in the small data setting and produce empirically successful results. However, this is likely because of *how* the joint model is used during training, which does not require a perfect generative model of language (or language with respect to programs) for either amortizing inference or abstraction in order to use language as a heuristic during learning.

A full definition of the statistical translation model we use can be found in (Gal & Blunsom, 2013). We re-summarize important details here. The IBM family of translation models estimates the conditional token-token probabilities $\mathrm{P}_{\mathcal{T}}[w|l]$ on the basis of *alignment* variables $a_{l,d}$, which specify a direct correspondence between tokens in parallel texts (e.g. a word in a task description and a program primitive.) These alignments are *many:many* between tokens in programs and natural language sentences – a given word can correspond to multiple primitives, and vice versa. Conditioned on a set of *alignments* from paired programs and descriptions, the conditional probabilities in *both* directions (the probability of generating a program primitive in a program based on the presence of a word in a sentence, and vice versa) are defined by marginalizing over the alignment variables. We provide one direction ($\mathrm{P}_{\mathcal{T}}[w|l]$), as the other is symmetrical:

$$\mathrm{P}_{\mathcal{T}}[w|l] \propto \sum_{a_1} ... \sum_{a_m} \mathrm{P}[w, a_1...a_m|l] \propto \prod_{i=1}^{m} q(a_i|i, l, m)$$

where a_i are alignment variables inferred over a paired corpus and $q(j|i, l, m)$ can be interpreted as the probability of alignment variable a_i (for the token with index i in a program) taking value j (where j is an index into the corresponding sentence) conditioned on the lengths l and m of the program and natural language sentence (Gal & Blunsom, 2013).

These alignments are inferred by approximately inverting the generative model in (Gal & Blunsom, 2013) to maxi-

mize the likelihood of the observed paired sentences and programs. One implementation detail: the alignment algorithm operates over pairs of strings. For convenience we infer alignments between sentences and linearized token sequences in the program tree (which can be done with complete recoverability of the original program tree (Andreas et al., 2013)). This is another inductive assumption that we choose after preliminary experimentation and find that our implementation yields strong empirical results regardless.

The IBM translation model is a noisy-channel generative model that requires an additional language model $p(d)$ to generate language (Gal & Blunsom, 2013; Heafield, 2011). We use an efficient parallelized implementation for inferring the translation model parameters from (Koehn et al., 2007), which also contains a basic language model inference algorithm inferred over the full corpus of training task sentences (as a trigram model, which we again find simple but effective for our very small data setting). Specific model hyperparameters for all experiments are available in the released code repo (in the experiment runtime commands.)

Mutual exclusivity: Section 5.1 of the main paper also describes how the joint model can be modified to include language-specific priors, such as a simple implementation of the well-known **mutual exclusivity** prior documented in the cognitive language-learning literature (Markman & Wachtel, 1988; Gandhi & Lake, 2019) and given a Bayesian formulation in (Frank et al., 2009). We provide an implementation to demonstrate that the joint model can be easily extended: specifically, a simple mutual exclusivity assumption can be added into the joint model by simply updating the compositional translation model to include additional distributions $t_{ME}(d_{new}|l)$ where d_{new} are words that *only* appear in unsolved training tasks and

$$t_{ME}(d_{new}|l) \propto \alpha \mathrm{P}[l|\mathcal{L}, \theta_{\mathcal{L}}]^{-1}$$

new words are now assumed to correspond to primitives *inversely* proportional to their current usage under the learned program prior. As we show in the next section, incorporating this prior at the level of the joint model can be used to approximate mutual exclusivity assumptions in the learned search heuristic, encouraging exploration in the presence of new words.

Practically, we calculate the mutual exclusivity prior in our concrete implementation by leveraging the *alignments* upon which our token-token translation probabilities are defined. Specifically, we add *pseudoalignments* between each d_{new} and each $l \propto \alpha \mathrm{P}[l|\mathcal{L}, \theta_{\mathcal{L}}]^{-1}$; when the token-token translation probabilities marginalize over the latent alignments and these pseudo alignments, the resulting translation probabilities encode the mutual exclusivity prior.

S5.2 Integrating the joint model into amortized conditional search

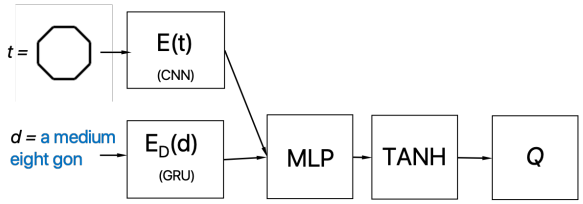

Figure 2. Architecture of the language-conditioned neural model $Q(\rho|d,t)$. The model takes as input task examples t. These are encoded using a domain-specific encoder $E(t)$. The model additionally takes in task descriptions d, encoded using a languag encoder $E_D(t)$ (implemented as a GRU). Task encodings are concatendated and feed to an MLP and activation layer and output a tensor Q. This parameterizes a distribution over program bigrams in the final DSL, which defines a conditional distribution from which to enumerate programs during search.

The amortized conditional inference model $Q(\rho|t)$ (Sec. 4.2) extends straightforwardly in LAPS to condition on language $Q(\rho|d,t)$ (Sec. 5.2). Importantly, the training procedure in Sec. 4.2 (training the neural model on samples from the prior) also extends to the language-enriched condition (training the neural model on samples from the joint prior, which include generated language annotations.)

In our experiments we implement the concrete neural model $Q(\rho|d,t)$ in our experiments by extending modularly on the original model in (Ellis et al., 2021) (and in the supplemental S4.2) for direct comparison. Our full architecture therefore has *three* modular components to additionally condition on language:

1. A *natural language task descriptions encoder* $E_D(d)$. This receives the task description d as input. We implement this as an RNN model using a bidirectional GRU (Cho et al., 2014) with 64 hidden units; we embed natural language symbols as 64-dimensional vectors, and randomly initialize and backpropagate through the embedding during training. We tokenize the sentences in u on whitespace and concatenate each sentence, delimited by special start and end of sentence tokens. At test time, we replace any OOV tokens with a special UNK token.

2. A domain-specific task encoder $E(t)$, following S4.2.

3. A bigram transition model over program primitives, following S4.2. To condition jointly on $E_D(d)$ and $E(t)$ we simply concatenate these two embeddings and update the first layer of the MLP to take the 128-dimensional concatenated embeddings as input.

5.3 Abstraction learning as joint model compression

Finally, the *abstraction learning* model in (Ellis et al., 2021) can also be generalized to condition on language, by extending the optimal library inference algorithm with respect to the program prior to an optimal library inference algorithm with respect to the joint model over language and programs (Eq. 6 and 7, main text.)

In our concrete implementation with respect to the DreamCoder algorithm, this means extending the description-length compression objective – originally defined over the program library and training task programs – to include the translation model definition. The main paper defines a description-length prior over the compositional translation model (Eq. 10). Optimizing this tractably requires redefining the abstraction algorithm in (Ellis et al., 2021) – which refactors λ-calculus programs via $lambda$-abstraction (see S4.3 for a summary) – to also jointly re-estimate the description length of the translation model $\mathcal{T}(d_t|\rho, \mathcal{L}')$ using the refactored programs under the new candidate library $\mathcal{L}'$.

We implement an efficient approximation that can be calculated with respect to the classical statistical translation model described in S4.1 (Gal & Blunsom, 2013). In particular, we leverage the *alignment*-based definition (which uses latent correspondences inferred between program tokens and sentence tokens in paired programs and descriptions) to approximate $-H(\mathrm{P}_{\mathcal{T}}[w|l]) = -\log(\mathrm{P}_{\mathcal{T}}[w|l])$, the entropy of the token-token translation probabilities.

Specifically, as the IBM model defines the conditional token-token probabilities

$$\mathrm{P}_{\mathcal{T}}[w|l] \propto \sum_{a_1} ... \sum_{a_m} \mathrm{P}[w, a_1...a_m|l]$$

marginalized over alignments, where (slightly abusing notation) in any given paired program and sentence description we will have estimated a set of alignments $a_{w_j,l_k...l_n}$ between the j-th token in the description corresponding to one *or more* tokens $l_k...l_n$ in the paired program. We therefore define the *description*-length of each token-token translation as the sum of the description lengths of the alignments which express it under a library $\mathcal{L}$:

$$\sum_{a_i} ... \sum_{a_m} \mathrm{P}[d, a_1...a_m|l, \mathcal{L}] \propto \sum_{a_1} ... \sum_{a_m} |a_i|_{\mathcal{L}}$$

and the description lengths under the *refactored* library $\mathcal{L}'$ containing new abstractions compresses according to

$$|a'_{w_j,l'_k...l'_n}|_{\mathcal{L}'} < |a'_{w_j,l_k...l_n}|_{\mathcal{L}} \iff \{l'_i \text{contains only } l_k...l_n \text{ as subcomponents} | l'_k...l'_n\} \quad (3)$$

and we say that a primitive $l \in \mathcal{L}$ is a *subcomponent* of a refactored abstraction $l \in \mathcal{L}$ if the abstraction can be β-reduced such that l appears in it. That is, a refactored alignment $a' : w_i \to \{l'...l_n\}$ is compressed only when a new abstraction l' encapsulates over a strict subset of the constituent program primitives already aligned to the word in the original alignment. This allows us to re-approximate the description length of the new translation model with respect to a semantically-equivalent program refactoring without inducing $\mathrm{P}_{\mathcal{T}}[w|l]$ from scratch (which would require retraining the full translation model over the sentences and refactored programs.)

S6. Experiments

This section describes additional details on each of the domains – *string editing, compositional graphics,* and *scene understanding* – in Section 6 of the main paper (see **Figure 2, main text** for examples from all three domains, shown along with the synthetic and human language annotations). We also provide additional details on the model and baseline hyperparameters available for each domain. All datasets generated for these experiments (including human language annotations) are released and links to static repositories are provided in the code release. We also release a complete set of commands to exactly replicate all model experiments.

All experiments for were conducted on a high-powered computing cluster using a fixed training budget of wall-clock search time per task for all models and baselines in a given domain (determined via hyperparameter search using the baseline model per domain, and reported on a per-domain basis below). The experiments on the string editing and graphics domains used models trained using 48 CPUs for search (using the original parallel enumerative search implemented in the released code for the DreamCoder model in (Ellis et al., 2021)); and the experiments trained on the scene reasoning task used 24 CPUs (as preliminary experiments revealed that these experiments required shorter search time for our main model, and we wished to reduce the carbon footprint of the remaining experiments after our first two domains.)

For all experiments we train the neural models for 1×10^4 gradient steps. For experiments with language-guided compression, we use an upper bound of 5 new abstractions introduced per iteration. For mutual exclusivity experiments, we set $\alpha_{ME} = 0.1$. For all experiments, during program-only compression (see (Ellis et al., 2021) for a discussion of program-only compression hyperparameters) we use the hyperparameters from (Ellis et al., 2021) for parsimony with earlier work: a structure penalty of 1.5 and pseudocounts = 30.

S6.1 Domains

(See **Figure 2**, main text for examples from all three domains, shown along with the synthetic and human language annotations.) As discussed in the main paper, each domain consists of a dataset of *tasks*; a set of procedurally generated *synthetic language annotations*; and a set of *human language annotations* provided by Mechanical Turk workers; we also described the *base primitives* $\mathcal{L}_0$ with which all models (including baselines and ablations) were initialized for each domain.

S6.1.1 String Editing

Tasks: structured string transformation problems taken from a publicly released dataset in (Andreas et al., 2017) (n=1000 train; n=500 test). Tasks consist of input dictionary strings transformed using randomly sampled regular expression transducer (n=30 examples per task). Transducers were sampled according to abstract templates defined in (Andreas et al., 2017) and required identifying matched sequences of characters and *adding* letters before them; *removing* sequences; *replacing* them with new sequences, or *doubling* the sequence each time they appeared (See **Figure 2A, main text**).

Language data: The human language dataset for this domain was previously collected by (Andreas et al., 2017). We defined a synthetic grammar of high-level templates over the ground truth regular expression transducers (corresponding to the original templates used to generate the tasks.) The synthetic templates were defined based on language from the original human annotations, and in most cases closely matched the true human provided annotations (which were generally quite structured), though with significantly less variation (the original language contained multiple human descriptions per task. We generate a single synthetic for each one. The synthetic dataset has a vocabulary size of n=44 for both train and test. We use the human annotations in the original dataset when evaluating on human data, which have a vocabulary of n=727 (train) and n=622 (test).) We generate a synthetic dataset on this domain partly because of inaccuracies noted in (Andreas et al., 2017). The released code contains the complete generation procedure for these synthetic annotations. See Figure 2A for representative tasks with examples, synthetic language, and human descriptions.

Initial program primitives: We initialize all models with a set $\mathcal{L}_0$ of LISP-like primitives that operate over substring sequences to both construct regular expression match sequences and manipulate strings, augmented with three text manipulation-specific primitives intended for executing constructed regular expression sequences; `t` is a polymorphic type variable using standard Hindley-Milner polymorphism typing (Pierce, 2002). The execution engine does include a regex-matching model; however, the synthesis model is naive to this execution engine and simply searches for manipulations over the input strings and the regexes as data arrays.

$\mathcal{L}_0$ contains 14 substring manipulation primitives, given below with type information. We also give a semantic gloss for primitives that are not standard LISP primitives.

- `if (bool → t → t → t)`
- `cons (t → list(t) → list(t))`
- `car (list(t) → t)`
- `cdr list(t) → list(t`
- `map` $((t_0 \rightarrow t_1) \rightarrow list(t_0) \rightarrow list(t_1))$
- `tail (list(t) → t)`
- `append (t → list(t) → list(t))`
  Appends element to end of list.
- `revcdr (list(t) → list(t))`
  Takes all except the last element of the list.
- `match (substr → substr → bool)`
  Returns true if the first argument, when executed as a regular expression, matches the second argument.
- `regexsplit (substr → fullstr → list(substr))`
  Attempts to execute the first argument as a regular expression, and splits the second argument into a list of substrings, using the regular expression match as a delimiter (and includes the matched sequences in the returned list.)
- `flatten (list(substr) → fullstr)`
  Flattens a list of substrings back into a string.
- `rconcat (substr → substr → substr)`
  Concatenates two substrings.
- `rnot (substr → substr)`
  Takes a substring argument `s` and returns the substring literal [^ s]
- `ror (substr → substr → substr)`
  Takes substring literals a and b and returns the substring literal ((a)—(b))

We also include 26 character constants of type `substr` and constants `dot` (regular expression wildcard character) and `empty` (empty string).

Domain hyperparameters We largely follow prior work (Ellis et al., 2021) to set algorithm training parameters; the

earlier (Ellis et al., 2021) uses a 720s enumerative search budget for solving both text editing and general list manipulation tasks. We use the same 720s enumerative budget here.

The encoder E(t) follows the domain-specific encoder used for text and list editing problems in (Ellis et al., 2021), a 2-layer GRU with 64 hidden units. The model is trained for a fixed gradient step budget (10,000 gradient steps) and we sample equally at random between supervision on the solved training tasks (and their solution programs in the current DSL) and samples from the joint generative model. As with (Ellis et al., 2021), when generating tasks from the generative model, we use randomly sample inputs (on which we execute generated programs to produce an output.)

S6.1.2 Compositional Graphics

Tasks: inverse graphics problems (n=200 train; n=111 test) where each synthesis problem is specified by an image and solved by synthesizing a program in LOGO Turtle graphics (Abelson & DiSessa, 1986). The domain is inspired by the graphics domain in (Ellis et al., 2021) but intentionally re-designed to be much more challenging (ground-truth programs are much longer on average in the base programming language) and explicitly compositional: the training and testing tasks contain *simple shape tasks* defined by compositional parameters for a set of basic shapes (*a small triangle, a medium square; a small semicircle*); *complex shape tasks* that require inferring more challenging (and longer) parameterized shapes (*a greek spiral with eight turns*); and *compositional tasks* defined by geometric rules and relations over the simple shapes (*a seven sided snowflake with a short line and a small triangle as arms; a small triangle connected by a big space from a small circle*) (See **Figure 2C**).

Simple parameterized shapes are either polygons (*triangle, square, [n] gon*), curves (*semicircle, circle*) or *line*s. Simple shapes are parameterized by one of three sizes (*small* or *short*; *medium*; and *big*). When generating synthetic language descriptions, pluralized objects are tokenized with separate tokens for the noun lemma and a token for the plural suffix (e.g. *square s*).

Complex parameterized shapes require constructing more complex images out of basic lines, and are intended to evaluate performance on tasks that pose a greater search challenge in the initial DSL, and whose structure is not directly cued by compositional relationships over easier components. Further, the complex shapes can be solved using abstractions (e.g. for repeatedly rotating a pen at right angles) that are not directly cued by shared lexical names – we evaluate the algorithm's ability to learn and use abstractions that correspond to useful sublexical structures shared across multiple lexemes. We define four template families for complex shapes: *spiral*s, *staircase*s, *zigzag*s, and *star*s.

Compositional graphics tasks invoke compositional relationships over the simple parameterized shapes. We define templates for generating 6 families of compositional tasks: *nested*, *next to*, *separated by*, *connected by*, *in a row*, and *snowflake*s.

Language data: We gather human language annotations by asking Mechanical Turk workers to write an image description for the rendered graphics images that specify each task. Each worker labeled 20 training and 10 testing images after viewing a disjoint, randomly sampled set of 15 example images paired with their synthetic language captions. (Workers were asked to write a *short, clear description that a person or robot could use to recreate the picture*, and told that the examples were paired with *automatically generated captions as an example of the kinds of descriptions you could write for this picture*.) We control for description quality by requiring workers to complete a reference task on their own descriptions: after writing their initial annotations, workers were required to correctly match each annotation to the target image (from amidst a set of 12 distractors drawn heuristically from similar images on the full task dataset, and other images they themselves had described), and only annotations correctly matched to the target image were retained (workers were given a chance to redescribe pictures they failed to match to their own captions.) We preprocess the human dataset minimally to standardize number terms (e.g. we use the same token type for both *3* and *three*) and to split plurals into a lemma and suffix, as in the synthetic dataset. The final dataset has a vocabulary size of n=562 for both train and test.

As with the string editing domain, we define a synthetic dataset using parameterized templates based on systematic language reused in the human annotations (see Figure 2A for a comparison between human annotations and synthetic language); as with that domain, we choose a synthetic dataset to ensure systematic re-use of high level terms for repeated compositional objects (such as the "n-gon" or "snowflake" terminology.)

We then generate graphics tasks by defining parameterized templates over ground truth programs *in* $\mathcal{L}_0$, and a corresponding generator for synthesizing natural language descriptions based on each ground truth program. It is important to note that the templates are defined at any extremely high level and were written with respect to low-level programs in a simple graphics language (many of which were derived by generalizing compositionally over complex structures in (Ellis et al., 2021), such as the 'snowflake' images).

Initial program primitives: For comparison with prior work, our initial library on this domain (and the base language used to generate the ground truth graphics programs) is an implementation of the LOGO Graphics DSL used in (Ellis et al., 2021), which consists of four typed, impera-

tive primitives modeled within the $\lambda-$calculus with a state monad S:

```
move:  distance → angle → S → S
pen-up:  (S → S) → S → S
for:  int → (S → S) → S → S
get/set:  (S → S) → S → S
```

as well as four arithmetic operators (+, -, *. /), integer constants (1-9), unit distances and angles (1 meter and 2π radians), and special values ∞ and ϵ.

Figure 3 (main text) shows examples of the graphics tasks, synthetic descriptions, human descriptions, and sample programs in the ground truth initial DSL.

Domain hyperparameters We largely follow prior work (Ellis et al., 2021) to set algorithm training parameters. Consistent with the graphics program experiments in (Ellis et al., 2021), we train all models, including baselines and ablations, using an enumerative search budget of 1800s per task (both when using pure enumerative search from the DSL prior, and neurally-guided search conditioned on the task examples and language descriptions); the results in Table 1 compare the relative advantage of our model given this fixed search time. We train all models on 48 CPUs during parallel enumerative search, and run the algorithm for a maximum of 27 iterations (see learning curves. As we run multiple random seed replications of models in this domain, we tuned the iteration limit based on performance on the first replication, allowing models models to train while performance continued to increase. To conserve computational resources, we later stopped several of our own model replications before 27 iterations, as they had reached near ceiling performance. As we report the best held-out test score across all 27 iterations for any one model, the early stopping would only serve to give a conservative estimate on performance for these models.) We randomly reorder the training set of tasks once before the first loop, then iterate through batches of n=40 tasks at each iteration; learning curves show results from evaluating on held-out tasks every n=3 iterations.

The encoder E(t) follows the domain-specific encoder used for the original graphics domain in (Ellis et al., 2021) for a more direct comparison: we use a 6-layer CNN, where each layer consists of a 64x64 2D convolutional sublayer with kernel size = 3, a RELU activation sublayer, and a max-pooling sublayer with kernel size = 2. The model is trained for a fixed gradient step budget (10,000 gradient steps) and we sample equally at random between supervision on the solved training tasks (and their solution programs in the current DSL) and samples from the joint generative model.

S6.1.3 Scene Reasoning

Tasks: inductive scene reasoning tasks (n= 212 train; n=115 test) where each synthesis problem is specified by a structured input scene, and outputs can be a number (*how many red rubber things are there?*), a boolean value (*are there more blue things than green things?*), or another scene (*what if all of the red things turned blue?*). This domain is modeled on CLEVR (Johnson et al., 2017) but designed to support non-linguistic, inductive synthesis in the programming-by-example paradigm: each task is specified with *n=7* paired input output examples. See **Figure 2B, main text** for example tasks showcasing the original and extended templates, synthetic language annotations, and human language annotations.

The dataset includes questions randomly generated from the following subset of the *original CLEVR question templates* (see (Johnson et al., 2017) for additional details on the task generation process and question templates; we also release our own augmented question generation code and the full dataset):

- **zero_hop**: questions that require counting or answering an attribute query about a subset of objects in the scene. (e.g. *How many small cylinders are there?*; *What material is the purple thing?*).
- **one_hop**: questions similar to the *zero_hop* tasks, but that require reasoning over an additional relational query (e.g *What number of things are right the small gray thing?*).
- **single_or**: questions that additionally introduce a *disjunction* between sets of objects. (e.g. *How many objects are either large metal spheres or large rubber things?*)).
- **(compare_integer**: questions that additionally introduce a $\geq$ or $\leq$ operator between counts of sets of objects. (e.g. *Is the number of large rubber cubes less than the number of large green rubber things?*)
- **same_relate**: questions that additionally require reasoning about other objects with the same attribute as a specified object. (e.g. *How many other things are there of the same size as the cyan thing?*).

We choose these templates as a representative subset of the style of the full CLEVR dataset, that requires the full language of high-level primitives in (Johnson et al., 2017) to solve. We omit some longer questions in the same format (e.g. *two_hop*) as our intention is to compare synthesis baselines, rather than to achieve SOTA performance on CLEVR: this would likely only increase the computing resources needed to compare the various methods and we

already found a significant differential between our model and the baselines on the shorter questions.)

We also add *new* question templates generated in the style of the original CLEVR tasks, but designed to model other common AI tasks (such as generating new scenes based on existing ones) and to require new abstractions (that were not expressible in the original restricted symbolic language used to generate scenes in (Johnson et al., 2017)):

- **localization**: questions for object localization. These return an output *scene* consisting of a localized set of objects based on a set of query attributes (e.g. *Find the gray rubber thing.*).
- **remove**: questions that either return an output *scene* with a subset of the objects removed, or that query about latent scenes where a subset of objects has bee removed. (e.g *What if you removed all of the gray metal things?*; *If you removed the green cubes, how many cubes would be left?*).
- **transform**: questions that either return an output *scene* where a subset of the objects has been *transformed* to set new attributes, or that query about latent scenes where a subset of objects has been modified this way. (e.g *What if all the blue metal things became rubber things?*; *If all of the large yellow rubber things became gray spheres, how many gray spheres would there be?*).

We treat these as program synthesis tasks: the input scenes are specified as *symbolic scene graphs consisting of an array of structured, objects defined as a dictionary of their attributes*, and programs are designed to manipulate these structured arrays (this data structure is the original format in which scenes themselves are generated in (Johnson et al., 2017); the images displayed in Figure 3, main text are rendered using the original image rendering pipeline). Our intention is *not* to build a visual reasoning architecture: rather, we are interested in learning structured manipulations of scenes. We see work in *inverse graphics* (such as (Yi et al., 2018)) which outputs a structured scene graph based on pixel images as the *first* step in a symbolic processing and reasoning pipeline as analogous; we are interested in the structured manipulation of these scene representations.

Language data: Synthetic language annotations are generated based on the original high-level templates in (Johnson et al., 2017), as well as additional templates we define for the extended questions in the same style. We gather human language annotations by asking Mechanical Turk workers to write an instruction or question describing the set of inductive examples. However, due to the difficulty of solving certain tasks in a limited time frame based on the inductive examples alone (such as the questions about disjunctions over scenes), we show Mechanical Turk workers the synthetic descriptions for this domain and ask them to write a semantically similar description that changes more than one word in the original caption, and that would be "more natural for a human to understand". This paraphrasing paradigm is similar to that used in (Wang et al., 2015), though we find that in comparison to other domains it generates less diverse language data.) We remove all punctuation, tokenize on spaces, and use an additional domain heuristic to stem all plurals (e.g. *cubes*).

Initial program primitives: We initialize all models with a set $\mathcal{L}_0$ of LISP-like primitives. These are similar to the initial list manipulation primitives used in the *string editing* domain: as both domains can be treated as manipulating structured arrays, we are interested in learning differentiated, domain-specific abstractions based on a very similar base language. $\mathcal{L}_0$ also includes primitives for querying attributes of objects on the domain (these are typed getters that simply query the object dictionary of attributes) and several domain-specific functions necessary for manipulating these attribute. We deliberately use a much more base level programming language than the high-level, domain-specific language hand-designed in (Johnson et al., 2017); our goal is to *learn* the necessary abstractions.

We give a semantic gloss for primitives that are not standard LISP primitives.

- `if (bool → t → t → t)`
- `cons (object → list(object) → list(object))`
- `car (list(object) → object)`
- `map` $((t_0 \to t_1) \to list(t_0) \to list(t_1))$
- `fold` $((list(t) \to list(t)) \to (t \to list(t) \to list(t)) \to list(t))$
- `len (list(t) → int)`
- `>` `(list(t) → bool)`
- `<` `(list(t) → bool)`
- `set_union (list(t) → list(t) → list(t))`
- `set_intersect (list(t) → list(t) → list(t))`
- `set_difference (list(t) → list(t) → list(t))`
- `relate (object → relation → list(t))` Returns an array of objects that satisfy a spatial relation with respect to an input object.

We also include *equality* comparators for each of the attribute types (e.g. `eq_color?`; *getters* for each attribute, and *setters* for each attribute. We also include integer constants 0-9 for counting and constants for the attributes (`blue, red, big, small, rubber, metal`) based on the original object and spatial relation constants (Johnson et al., 2017).

Domain hyperparameters: We run a coarse hyperparameter search based on the baseline model to set the domain hyperparameters. We train all models, including baselines and ablations, using an enumerative search budget of 1000s per task and run the models for a maximum of 5 iterations. we run multiple random seed replications reordering the training set, in the same way as the compositional graphics domain. The results in Table 1 also compare a *curriculum* ordering of the training set based on the number of tokens in the synthetic language captions (split on spaces.)

The encoder E(t) is a variant of the RNN-based domain-specific encoder used for text and list editing problems in (Ellis et al., 2021) (as well as the string editing domain). The model is trained for a fixed gradient step budget (10,000 gradient steps) and we sample equally at random between supervision on the solved training tasks (and their solution programs in the current DSL) and samples from the joint generative model. As with (Ellis et al., 2021), when generating tasks from the generative model, we use randomly sample inputs (on which we execute generated programs to produce an output.) We encode the symbolic scene data structures with the RNN by encoding a flattened version of the scene graph. The scene graph is originally stored as a dictionary of attributes; when flattened, we indicate the dictionary structure using special tokens to denote the keys and the start and end of any array delimiters (the original scene graph is fully reconstructable from the flattened version.)

S 6.2 Results and Additional Qualitative Results

In this section, we discuss additional qualitative results from an in depth exploration of the graphics domain that were omitted from the main paper for space, but provide additional insight on the behavior of the learned model in the hardest learning domain (based on the differential between baseline and LAPS-augmented performance.)

Learned abstractions and synthesized programs. Figure S4 (supplement) show sample abstractions in the final libraries $\mathcal{L}_f$ for the best performing models in the graphics domain as a concrete exemplar of abstractions that are learned and how they are used, along with sample tasks solved with these abstractions. The figures are shown as dependency graphs to indicate how progressively more complex abstractions *build* on abstractions at prior iterations of learning; we also show selected probabilities from the translation model (depicted are examples from the top-3 primitive translations for a given word; some primitives are not high probability translations for any word.)

Joint generative model samples. Figure S3 (supplement) shows samples from the joint generative model on the graphics domain (programs from the library which are executed to produce the task example image, and translated to produce language annotations) at early and later stages of training, indicating that the joint model itself improves as learning improves, which itself allows better training for the conditional inference model and better abstraction guiding based on language.

References

Abelson, H. and DiSessa, A. A. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press, 1986.

Andreas, J., Vlachos, A., and Clark, S. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 47–52, 2013.

Andreas, J., Klein, D., and Levine, S. Learning with latent language. *arXiv preprint arXiv:1711.00482*, 2017.

Artzi, Y., Das, D., and Petrov, S. Learning compact lexicons for ccg semantic parsing. 2014.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Dechter, E., Malmaud, J., Adams, R. P., and Tenenbaum, J. B. Bootstrap learning via modular concept discovery. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 990–998. JMLR. org, 2017.

Dumancić, S. and Cropper, A. Inventing abstractions by refactoring knowledge.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., and Tenenbaum, J. Learning libraries of subroutines for neurally–guided bayesian program induction. In *Advances in Neural Information Processing Systems*, pp. 7805–7815, 2018.

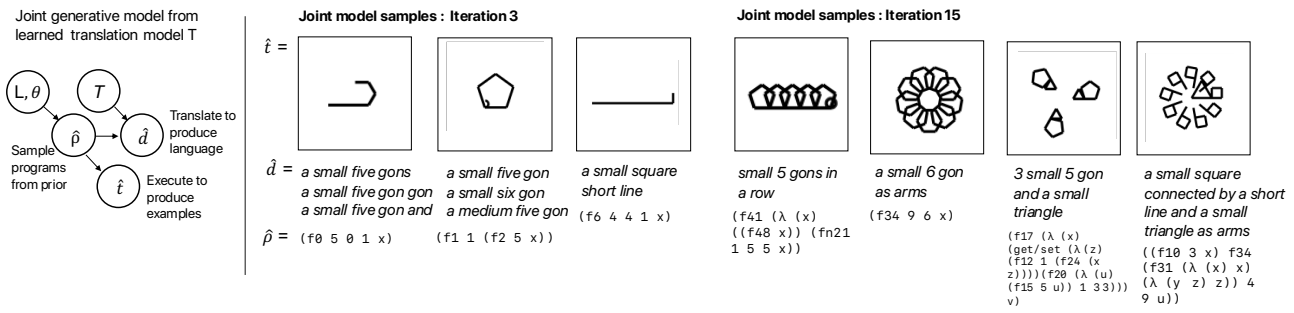

Figure 3. (left) Joint generative model J over programs sampled from the DSL prior and natural language produced by the translation model $T(D|\mathcal{L})$, inferred from solved training tasks. Samples from the model are used to train a neural synthesizer to guide search on more challenging, unsolved tasks. (right) Samples from the J generative model in the graphics domain shows how program complexity increases and generated language improves across iterations, as the system both adds richer abstractions to the DSL and learns better alignments over the solution set, enabling the trained neural model to solve more complex tasks

.

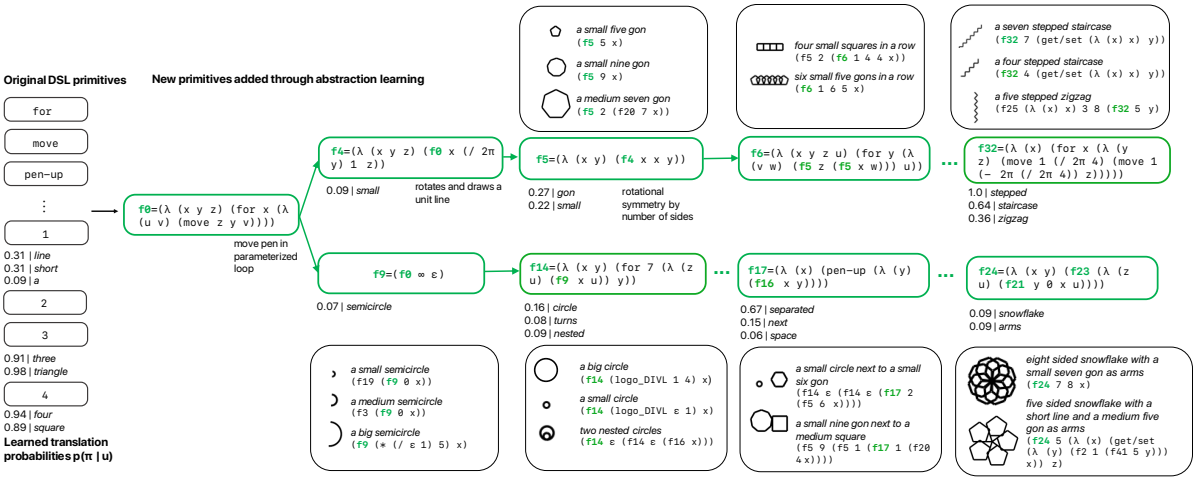

Figure 4. Abstractions and programs learned for the graphics domain. Sample abstractions (right) learned from a minimal starting DSL (left) for solving progressively more complex graphics program synthesis tasks with language annotations. Also shown with translation probabilities. Our iterative algorithm learns alignment-based translation probabilities between natural language words and program primitives to guide program search and abstraction (depicted are examples from the top-3 primitive translations for a given word; some primitives are not high probability translations for any word.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. Dreamcoder: Bootstrapping inductive program-synthesis with wake-sleep library learning. *PLDI 2021*, 2021.

Frank, M. C., Goodman, N. D., and Tenenbaum, J. B. Using speakers' referential intentions to model early cross-situational word learning. *Psychological science*, 20(5): 578–585, 2009.

Gal, Y. and Blunsom, P. A systematic bayesian treatment of the ibm alignment models. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 969–977, 2013.

Gandhi, K. and Lake, B. M. Mutual exclusivity as a challenge for deep neural networks. *arXiv preprint arXiv:1906.10197*, 2019.

Heafield, K. Kenlm: Faster and smaller language model queries. In *Proceedings of the sixth workshop on statistical machine translation*, pp. 187–197. Association for Computational Linguistics, 2011.

Johnson, J., Hariharan, B., Van Der Maaten, L., Hoffman, J., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2989–2998, 2017.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, pp. 177–180, 2007.

Lake, B. M., Salakhutdinov, R. R., and Tenenbaum, J. One-shot learning by inverting a compositional causal process. In *Advances in neural information processing systems*, pp. 2526–2534, 2013.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Liang, P., Jordan, M. I., and Klein, D. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 639–646, 2010.

Markman, E. M. and Wachtel, G. F. Children's use of mutual exclusivity to constrain the meanings of words. *Cognitive psychology*, 20(2):121–157, 1988.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*, 2019.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Pierce, B. C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Polosukhin, I. and Skidanov, A. Neural program search: Solving data processing tasks from description and examples. 2018.

Shin, E. C., Allamanis, M., Brockschmidt, M., and Polozov, A. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*, pp. 10824–10834, 2019.

Wang, Y., Berant, J., and Liang, P. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1332–1342, 2015.

Wong, Y. W. and Mooney, R. J. Learning for semantic parsing with statistical machine translation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pp. 439–446. Association for Computational Linguistics, 2006.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems*, pp. 1031–1042, 2018.

Zhang, Y., Pasupat, P., and Liang, P. Macro grammars and holistic triggering for efficient semantic parsing. *arXiv preprint arXiv:1707.07806*, 2017.