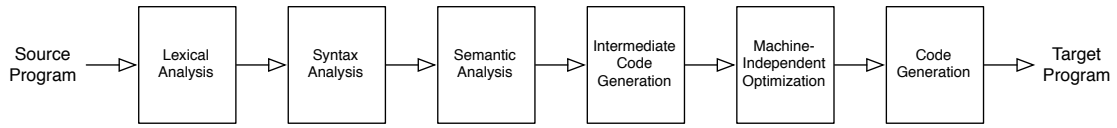
CS 301 Spring 2021 T4
Tiny Exploration 1
13 April

Tiny Compiler: Front End

1 Plan



To start the course we build a compiler that translates programs in the TINY calculator language to x86 machine code. This activity guides you to build the compiler's *front end*, including lexical analysis, syntax analysis, and semantic analysis. A later activity will build the *back end*: intermediate representation, optimization, and code generation.

1.1 Goals

☐ Preview course concepts by building a working compiler at accessible scale, with guidance. Concept exposure (not concept mastery) is the goal of this activity.

☐ Start learning Scala and associated tools.

☐ Try working with potential project teammates or tutorial group members.

1.2 Instructions

- **Complete this activity in groups of 3.** Choose teammates with whom you have not worked before. Include at least one CS 251 alum per group if possible.
- **Aim for a big-picture view of each compiler stage.** Do not worry if details are fuzzy: guess, follow intuition, experiment to see what works. Check in with me if you are lost or stumped.

2 Setup

Use a terminal in Linux, macOS, or WSL (Windows Subsystem for Linux).
WSL info: `https://cs.wellesley.edu/~cs301/s21/tools/#wsl`.

1. Install and configure IntelliJ, the Scala Plugin, and a Java 11 JDK with these steps:
   `https://cs.wellesley.edu/~cs301/s21/tools/#intellij-idea`
2. Clone the starter project:
   (a) Open IntelliJ and choose *New* > *Project from Version Control* from the menu.
   (b) Enter the URL `https://github.com/wellesleycs301s21t4/tiny-front.git`
   (c) Click *Clone*.
   (d) When prompted about whether to trust and open the BSP project, click *Trust Project*.
   (e) IntelliJ should now clone and automatically build the project. Wait for this to finish: watch progress bars at the bottom of the IntelliJ window.
3. Configure IntelliJ and a terminal session when IntelliJ has finished importing and building:
   (a) Open the IntelliJ preferences/settings from the menu (*IntelliJ* > *Preferences* or *File* > *Settings*, varies by OS).

i. In *Build, Execution, Deployment* > *Compiler*, enable *Build project automatically.*

ii. Close the preferences/settings.

(b) Open a terminal, `cd` to the `tiny-front` project directory if needed, and run the command `source env.sh`. This produces wrapper scripts for the TINY compiler and interpreter and put them on the PATH.

4. Find files in the upper left *Project* pane:

   (a) Find source code files in `tinyc/src/tiny`.

   (b) Find test files in `test`.

3 Implementation Language

This course uses Scala (`https://scala-lang.org`) as the main *implementation language* for all projects. Scala is a statically typed language with both functional and object-oriented features. It is typically compiled to run on the JVM (Java Virtual Machine) and is interoperable with Java. Compared to Java, it introduces several niceties and eliminates several annoyances.

CS 251 alums will see many features similar to Standard ML that are well-suited to many of the compiler tasks we will implement. If you have not had CS 251, the Java-like side of Scala should be quick to pick up. You will learn to use some of the other features over time. Do note that Scala is a large language with many parts; we typically stick to a relatively small subset of those parts that are clean and easy to comprehend.

Exercise 1. In the `tiny` project in IntelliJ IDEA, open several Scala files, skim some code, and note at least 5 substantive differences between Scala and Java.

Let's check in with the whole class here to share tips about Scala.

4 Compiler Architecture

Open `Compiler.scala`. This file contains the top-level logic of the TINY compiler. Each compiler stage referenced here is defined in a separate file of the same name. `AST.scala` and `IR.scala` define data structures to represent TINY programs in the compiler; the remaining files define functions that produce or consume these data structures.

Exercise 2. Read the compiler's `main` function. Take a look around. Skim header comments in each of the Scala files. Is this the compiler structure you expect? Note any surprises or questions.

Consider that programs are just data that are manipulated, computed upon, transformed, and so on. Like other data, program *representation* is independent from program *meaning*. This compiler implementation will deal with the same program represented as a source code string, a stream of lexical tokens, a tree capturing syntactic structure, graphs of control-flow structure, an assembly code string, and a machine code executable.

5 The TINY Language

Our *source language*, TINY[1], is a small calculator language. Informally, TINY *works as you expect*. Both the *syntax* – the structure of programs – and the *semantics* – the meaning of this structure – closely resemble those of many familiar programming languages. TINY includes: parenthesized addition expressions; an `input` expression that consumes and yields the next number value from user input; mutable variables; variable assignment statements; and print statements. Statements and expressions are evaluated eagerly left to right. To simplify parsing, all keywords and symbols (including parentheses) must be separated by spaces.

Exercise 3. The following TINY program takes two integer inputs i_1 and i_2 from the user and prints two numbers computed based on these inputs.

```
x = ( 4 + input ) ;
y = input ;
print ( ( 7 ) + ( y + y ) ) ;
print ( ( x + ( y + 3 ) ) + 97 ) ;
```

Execute this program manually with the inputs $i_1 = 6$ and $i_2 = 2$, in order. Show the values printed.

Exercise 4. Write a TINY program that takes 3 inputs i_1, i_2, and i_3 and prints the number equivalent to the arithmetic expression $2(i_1 + i_3) + i_2 + 7$. Also save it as: `src/test/tiny/ex4.tiny`

[1] TINY stands for *Treetop Investigators Needlessly Yammering*, *Truly Inimitable Northern Yew*, or whatever else you prefer

Asking you to execute and author TINY programs without much of a definition of the language is a bit devious, but your informed guesses are most likely correct. With an intuitive informal understanding of the language in hand, we now define the TINY syntax and semantics more formally.

6 TINY Syntax Definition

Syntax is the structure of a language. It determines what symbols or words are used to express programs in the language, but more importantly, how those symbols or words can fit together to build larger linguistic structures, much like we could define individual words, general parts of speech, and larger grammatical structures in human languages. Syntax does *not* define meaning of structures, *i.e.*, the computation they describe. That is the job of *semantics*, which we will cover later.

The syntax of TINY is given by the following *grammar*:

$$
\begin{aligned}
P &::= S^* \\
S &::= V \text{ `=' } E \text{ `;' } \mid \text{ `print' } E \text{ `;'} \\
E &::= N \mid V \mid \text{ `(' } E \text{ `+' } E \text{ `)' } \mid \text{ `input' } \mid \text{ `(' } E \text{ `)'} \\
V &::= [\texttt{a-zA-Z}]^+ \\
N &::= [\texttt{0-9}]^+
\end{aligned}
$$

A grammar describes the set of strings that are valid in a language by giving a structural definition in terms of two kinds of symbols:

1. *Terminal* symbols are concrete tokens appearing literally in strings of the language.

   Terminals are written in single-quoted `monospace font`, *e.g.*, '`print`'.

2. *Nonterminal* symbols represent abstract structures with one or more specific forms called *productions*. Each production is a sequence of symbols defining one form of the nonterminal more specifically.

   Nonterminals are written as upper-case letters in *italic serif font*, *e.g.*, "S". The productions of a nonterminal are given by writing the nonterminal symbol to the left of "::=" with its one or more productions to the right, separated by "|".

   We use a few shorthand notations in productions: the superscript symbol "*" indicates that the preceding item may appear zero or more times contiguously; superscript "+" (distinct from "`+`") indicates one or more; the notation "[`a-zA-Z`]" indicates any single letter; "[`0-9`]" indicates any single digit.

A grammar is a recursive definition. The TINY grammar is read as follows:

- A program, P, is a sequence of zero or more statements, S.
- A statement, S, is one of:
  - an assignment, composed of a variable name, V, followed by "` = `" followed by an expression, E, followed by "` ;`"; or
  - a print, composed of "`print` " followed by an expression, E, followed by "` ;`".
- An expression, E, is one of:
  - a number, N, composed of one or more digits;
  - a variable name, V, composed of one or more letters (but not the sequences `print` or `input`);
  - an addition expression, composed of "`(` " followed by an expression, E_1, followed by "` + `" followed by an expression, E_2, followed by "` )`";
  - a user input expression, composed of "`input`"; or
  - a parenthesized expression, composed of "`(` " followed by an expression, E, followed by "` )`".
- A variable name, V, is a sequence of one or more letters (except the sequences `input` or `print`).
- A natural number, N, is a sequence of one or more digits.

7 From Characters to Abstract Syntax Trees

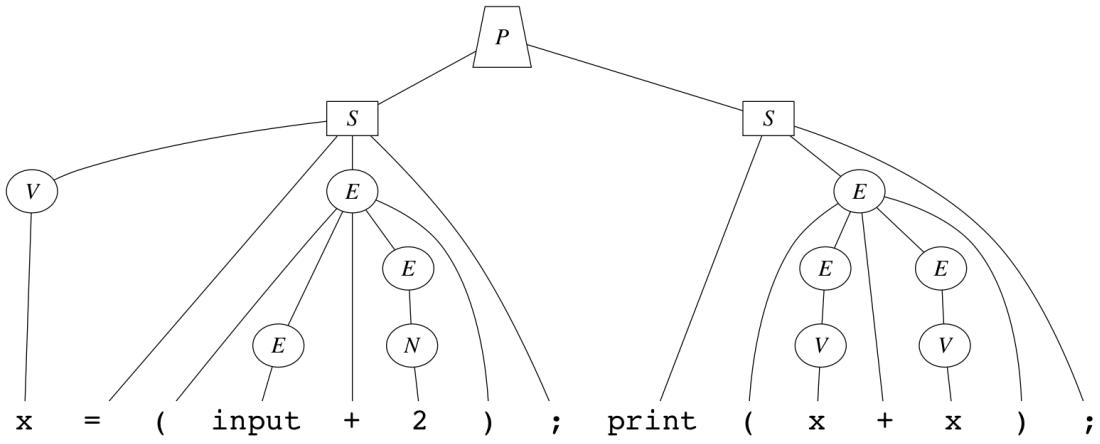
The first major step in a compiler is to extract the abstract syntax of the source program (*i.e.*, its larger hierarchical structure – a tree, in fact) from its concrete syntax (*i.e.*, the flat linear string of characters comprising the source code) according to the rules of the grammar. For simplicity, we break this process into two steps:

1. *Lexical analysis* or *scanning* groups individual characters into meaningful lexical tokens. (This will be the focus of our first tutorial meeting.) The syntax of TINY is designed to make lexical analysis trivial: all tokens are delineated by spaces.

2. *Syntactic analysis* or *parsing* uses the grammar to extract a hierarchical structure, called an *abstract syntax tree*, from the stream of lexical tokens.

7.1 Parsing Programs as Trees

The grammar allows us to understand the hierarchical syntactic structure encoded by a stream of tokens: a tree composed of nodes in the shapes given by productions of nonterminals in the grammar. The tree's inner nodes and root correspond to nonterminals in the grammar; productions determine the children of nodes; lexical tokens are leaves. For example, this tree shows the derivation of a program in the TINY grammar:

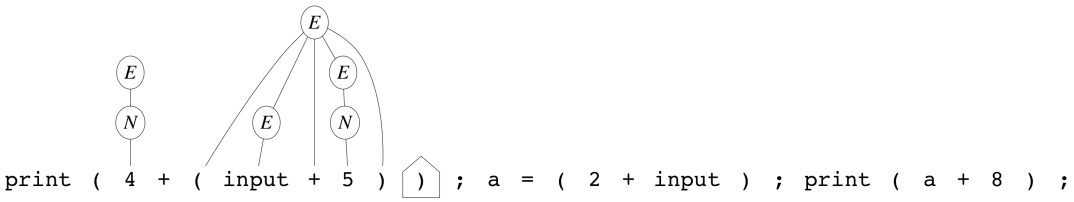


The process of recovering a syntax tree starting from only its leaves, the stream of lexical tokens, is called *parsing*. There are many ways to parse a stream of lexical tokens. We start with visual intuition for one *bottom-up* approach.

Informally, we read the stream of tokens from left to right, treating each token as a leaf and gradually building a tree above the leaves. Based on the next token and the trees constructed so far, we may: *shift* the token from unconsumed input to become its own tiny tree in the construction area; or *reduce* one or more trees on the right end to become children of a new larger tree. Eventually, this should yield a single P tree.

Some subtrees can appear in multiple contexts. For example, variable names appear both on the left hand side of assignment statements as well as in expressions. Informally, we decide how to treat them by understanding what we expect to see next. For example, an expression can never appear at the start of a statement in TINY. While it makes sense to reduce the first `x` to a V tree, it would not make sense to reduce it to an E tree. For now, we follow intuition rather than developing a precise algorithm for this style of parsing. Our second and third tutorial meetings explore top-down and bottom-up parsing in detail.

Exercise 5. Complete the parse tree for the following TINY program step by step from left to right. The next unconsumed token is shown in a pentagon. The subtrees we have so far are: `print`, `(`, E, `+`, and E.

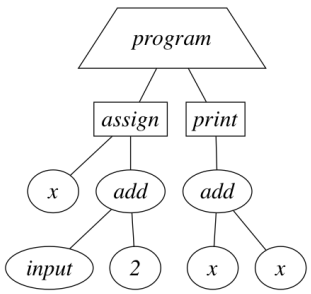


7.2 Abstract Syntax Trees

The parse tree shown in Section 7.1 makes the program's concrete syntactical structure clearer. While all tokens must be considered during parsing, many become superfluous once parsing is complete. At this point, we care only that the program (P) has the following structure:

- The first statement (S) is an assignment statement where:
  - The variable (V) has variable name `x`.
  - The expression (E) is an addition expression where:
    - The left expression (E) is an `input` expression.
    - The right expression (E) is a literal number (N) where the number value is `2`.
- The second statement (S) is a print statement where:
  - The expression (E) is an addition expression where:
    - The left expression (E) is a variable (V) where the variable name is `x`.
    - The right expression (E) is a variable (V) where the variable name is `x`.

We could draw this structure more simply as an *abstract syntax tree* (AST), collapsing unimportant details of the grammar derivation, and omitting the *concrete syntax* that makes it feasible to write programs as text.



Exercise 6. Draw an AST for the program you parsed in Exercise 5.

7.3 AST Representation

Before implementing a parser, we need a target representation for abstract syntax trees (ASTs). The file `AST.scala` defines the abstract syntax of TINY as a hierarchy of *case classes* that can be assembled to create a data structure representing a TINY program. `Seq` is a generic ordered sequence collection type, so `Seq[Stmt]` is the type of an ordered sequence of `Stmt` elements.

Exercise 7. Sketch the class hierarchy defined in `AST.scala`. How does it relate to the TINY grammar?

Scala's case classes support constructing instances without the `new` keyword. (For CS 251 alums, case classes are much like constructors of ML algebraic datatypes. They support pattern-matching.) To construct a `print` statement carrying the expression `301`, we just write `Print(Num(301))`, which results in a `Print` object whose `expr` is a `Num` object whose `value` is `301`.

Exercise 8. Using the AST types defined in `AST.scala`, write a Scala expression that constructs a representation of the AST you drew in Exercise 6 for the program from Exercise 5.

8 Implementing Parsing

Now it is time to implement what you have learned so far. Exercises in this section guide you through completing a parser for the TINY compiler starting from a partially implementation in `Parse.scala`.

8.1 Scanner (Lexer)

For lexical analysis (a.k.a. scanning), we use a `java.util.Scanner` to consume the program source code and break it up into lexical tokens. The `Scanner` has a few methods we use:

- `hasNext(p)` takes a `String` describing a simple pattern. Its return value indicates whether or not the pattern `p` describes the next token in the source code input. These patterns are *regular expressions*, which you may have used before. More in the upcoming assignment!
- `next(p)` consumes the next token, using the pattern `p` to define what part of the input comprises the token, and returns that token. `next(b)` assumes `hasNext(p)` is true (so always check first).
- There are also integer-specific versions `hasNextInt()` and `nextInt()` that check for and get the next integer encoded by the source code, respectively.

By default, `Scanner` distinguishes the boundaries of tokens by whitespace, matching the syntax of TINY. When specifying patterns for tokens, some characters have special meaning. Parentheses, plus, and star must be escaped with `"\\"` if you want the literal character: `"\\("`, `"\\+"`, `"\\*"`.

Exercise 9. Skim `Parse.scala` to see how provided code uses the `Scanner`. What pattern gets used for recognizing variable-name tokens?

Next week, the first tutorial readings, assignment, and meetings explore the principles behind the `Scanner` as well as more sophisticated techniques for lexical analysis.

8.2 Recursive Descent Parser

`Parse.scala` has the framework of a *recursive descent parser*, one of the more intuitive ways to structure a manually implemented parser for a simple grammar. The provided parser handles a subset of the TINY language: assignment statements, integer literal expressions, and input expressions. Your job is to extend it to handle the other types of statements and expressions.

Exercise 10. Write a TINY program using the subset of the TINY language supported by the provided parser and run the compiler. It should print the AST it has parsed. (Currently, it does nothing more.)

IntelliJ IDEA compiles the Scala code (*i.e.*, compile the compiler...) incrementally each time you save. To run the compiler on a test program in `src/test/tiny/simple.tiny`:

```
cd tiny-front
./tinyc src/test/tiny/simple.tiny
```

Section 7.1 parsed TINY programs by hand using a *bottom-up* approach: we let the tokens drive parsing and gradually build the parse tree from the leaves to the root. In contrast, recursive descent parsing is a *top-down* approach: when parsing a program, the parser knows that a program is composed of statements, so it first tries to parse a statement; when parsing a statement, the parser knows there are two kinds of statements, so it examines the first available token to distinguish which kind of statement to parse, then parses its parts; and so on. Reading the parser code is the most effective way to understand this model.

Exercise 11. Read `Parse.scala`. Consider only the subset of the grammar corresponding to the cases the parser support so far. How do the provided functions in `Parse.scala` correspond to the grammar for TINY? How do the different branches in each method correspond to the grammar? **Make sure to understand these points before starting the following exercises.**

Exercise 12. Extend the parser to handle `print` statements in `parseStmt`. Write a couple simple programs to test your extension. Why must the parser check for the *print* case first before the considering the *assignment* case (and not the opposite)?

Exercise 13. Extend the parser to handle parenthesized addition expressions in `parseExpr`. Write a few simple programs to test your extension. Hint: this should finally make your recursive descent parser live up to its name. How is the parser's use of recursion related to the grammar?

Exercise 14. (Optional) Extend the parser to handle "extra parentheses" expressions (the last production of E in the grammar) in `parseExpr`. Before coding, think carefully about how you will distinguish these from addition expressions. Write a few simple programs to test your extension.

9 TINY Semantics Definition

The *semantics* of a programming language describes what syntactic structures in the language *mean* or, in other words, how to evaluate them. So far, our definitions of the TINY semantics are vague, appealing to intuition and familiarity with other languages. We next explore a provided reference interpreter for TINY programs to understand TINY semantics more precisely. If you are curious about specifying the semantics of TINY formally, read the optional Appendix B.

9.1 TINY Reference Interpreter

An *interpreter* reads in a program's source code and *executes* or *evaluates* the program. An interpreter would be better named an *executor* or *evaluator*, just as a *compiler* would be better named a *translator*, but history chose the *interpreter* and *compiler* terms and they are well-established in common usage.

Why is a TINY interpreter relevant to our goal to build a TINY compiler? Defining how to interpret (execute/evaluate) an arbitrary program directly – *i.e.*, giving an *operational* semantics of the language – can often be more straightforward than defining how to compile (translate) an arbitrary program to another language – *i.e.*, giving a *denotational* semantics of the language. In preparation to build a TINY compiler, the TINY interpreter serves as a fairly precise, although informal, definition of the TINY language semantics. Later in the course (or optionally in Appendix B), we encounter more formal ways of specifying semantics.

Exercise 15. Read the file `Interpreter.scala`, which defines a full interpreter for the TINY language. How does the code structure relate to the AST data structure and the grammar of TINY?

Exercise 16. How does the interpreter track the values of variables as it evaluates a program?

Exercise 17. Briefly, summarize how the interpreter works overall.

Exercise 18. OK, we can't just let it sit there unused. Run the interpreter on some of your TINY programs. For example: `./tiny src/test/tiny/this-program-is.tiny`

10 Semantic Analysis

Even in a simple language like TINY, there exist syntactically well-formed programs that are not semantically valid. In other words, some programs *look* reasonable but have no reasonable meaning. The evaluation of such programs may – or in the case of TINY, definitely will – result in errors.

10.1 A Notion of Scope

In TINY, consider the issue of variable definitions and variable uses.

Exercise 19. Consider the following TINY program (also in `src/test/tiny/error.tiny`).

```
y = 7 ;
print y ;
print x ;
x = ( input + y ) ;
```

Is this program syntactically valid? (Does it parse?)
Does this program make sense intuitively?
How does it behave when evaluated with the TINY interpreter?

This program has a semantic problem: it attempts to use the value of variable `x` before it has defined a value for `x`. We *could* define that all variables are implicitly initialized to hold 0 at the start of the program. However, this is arguably not the most intuitive semantics and it is definitely not the semantics implemented by the TINY reference interpreter. (It raises a runtime error at the point where a program attempts to use an undefined variable.) In TINY, *variables must be defined before they can be used.*

Similar issues arise in most any programming language with names: a syntactically valid program may contain semantically invalid uses of names. More elaborate programming languages involve other kinds of semantic errors, such as type errors, array bounds errors, and more.

The *dynamic* mindset, which guides the design of languages like Python, tends to suggest that semantic errors should be raised only if the erroneous operation is actually encountered at run time. In contrast, the *static* mindset, which guides much of the design of languages like Scala, tends to suggest that potential semantic errors should be raised before the program ever runs and should, furthermore, prevent the program from executing at all, to avoid potentially encountering semantic errors during execution/evaluation.

Exercise 20. What are the strengths and weaknesses of the dynamic and static approaches toward semantic errors? Consider your own experiences. Consider different applications.

10.2 Resolving Names and Checking Definitions

The TINY interpreter takes a dynamic approach to handling semantic errors. The TINY compiler should take a static approach, introducing compile-time checks to ensure that programs use only defined variables. The variable-use checker or *scope checker* for TINY is quite straightforward thanks to the linear nature of TINY programs. (TINY has no control flow branching.)

Exercise 21. In `Compiler.scala`, uncomment the call to `ScopeCheck` in `frontend`. Complete the partially implemented scope checker in `ScopeCheck.scala`. Test your work on the broken program from Exercise 19. Note that `Set` has a `contains` method. You can report a scope error by `throw`-ing a `new ScopeError`.

11 Reflections

We have now defined the syntax and semantics of TINY, transformed a string of characters into a stream of lexical tokens, parsed the stream of lexical tokens to derive an abstract syntax tree, and analyzed the abstract syntax tree to determine if the program is semantically valid. We used several tools, ranging from our implementation language, Scala, to formal devices like grammars, ASTs, and operational semantics. The first half of the course (6 tutorial meetings) explores the topics in this activity in more depth.

Exercise 22. Note your favorite "aha!" moments, points of remaining confusion/curiosity, *etc.*

A Extending the Language

Exercise 23. (Optional) Extend the TINY Language with a multiplication expression. Syntactically, it is identical to the addition expression except that `+` is replaced by `*`. First consider how you will represent multiplication in the AST, then extend each phase of the compiler with support for multiplication.

Note that addition and multiplication will have many similarities. Simply adding a new `Times` case class may result in significant code duplication through the compiler where addition and multiplication may be treated similarly or identically. Although it will require changing some existing code, it may be beneficial to consider designing a `BinaryExpr` that carries two subexpressions and an operator.

B Big-Step Operational Semantics for TINY

Following the formal notation in this appendix is beneficial, but not required. If you get lost in the notation, do not spend too long in this section: read the interpreter (Section 9.1), then skip to Section 10.

An *operational* semantics defines a language's meaning by describing *how to evaluate programs* in terms of an abstract machine. A *big-step* semantics shows this evaluation all at once rather than step by step. Let's define syntax and evaluation rules for a TINY abstract machine.

B.1 Notation for a TINY Abstract Machine

Our abstract TINY machine consumes TINY programs written in concrete syntax. The rules of our machine use syntactic variables to extract subparts of TINY program syntax like a magical parser: The syntactic variable $\boldsymbol{p}$ represents a program (anything of the form P); $\boldsymbol{s}$ represents a statement (S); $\boldsymbol{e}$ represents an expression (E); $\boldsymbol{x}$ represents a variable name (V); $\boldsymbol{n}$ represents a number (N). We use these syntactic variables instead of the nonterminals E, *etc.*, since we will use them to capture a specific expression of the form E, rather than the class of all elements of the form E.

To model TINY variables, our machine tracks a *dynamic environment*, written *env*, that is a map from variable name to number value. Program evaluation begins with an empty environment, $\emptyset$. The following notations describe environment manipulations:

- $env[\boldsymbol{x} \mapsto \boldsymbol{n}]$ constructs a new environment identical to *env*, except that variable $\boldsymbol{x}$ maps to value $\boldsymbol{n}$.
- $env(\boldsymbol{x})$ yields the value, $\boldsymbol{n}$, to which $\boldsymbol{x}$ maps in *env*; it is valid only if such a mapping exists, *i.e.*, $\boldsymbol{x} \in env$.

To model user inputs to the program, our machine tracks a list of dynamic inputs, written *in*, which we assume is pre-populated with all inputs at the start of evaluation.[2] The syntax of an inputs list, *in*, is simply a space separated listed of numbers, *e.g.*, `240 251 301`. Outputs are produced in the same form.

B.2 Evaluation Rules for a TINY Abstract Machine

The evaluation rules in Figure 1 define how the TINY abstract machine works. The rules are organized under three *judgments*, each of which appears in a box. Each judgment has one or more *inference rules* that determine the conditions under which a judgment can be satisfied. Each rule is of the form:

$$\frac{\textsc{rule name}}{\textit{premise}_1 \qquad \ldots \qquad \textit{premise}_n}{\textit{conclusion}}$$

If all premises of a rule hold, then the conclusion holds. Within a rule, all occurrences of a given syntactic variables (such as *env* or $\boldsymbol{e}$) refer to the same syntactic structure.

[2] Actually, there's no reason that the implementation cannot prompt the user for these inputs interactively through the program execution. The way that the formal system underlying our operational semantics works, we can do the moral equivalent of guessing exactly the right inputs ahead of time automatically.

$\boxed{env, in, out, \boldsymbol{p} \downarrow out'}$ Evaluation rules for programs $\boldsymbol{p}$.

$$\textsc{empty}\ \frac{}{env, in, out, \downarrow out} \qquad \textsc{statement}\ \frac{env, in, out, \boldsymbol{s} \downarrow env', in', out' \qquad env', in', out', \boldsymbol{p} \downarrow out''}{env, in, out, \boldsymbol{s}\ \boldsymbol{p} \downarrow out''}$$

$\boxed{env, in, out, \boldsymbol{s} \downarrow env', in', out'}$ Evaluation rules for statements $\boldsymbol{s}$.

$$\textsc{assign}\ \frac{env, in, \boldsymbol{e} \downarrow in', \boldsymbol{n}}{env, in, out, \boldsymbol{x}\ \texttt{=}\ \boldsymbol{e}\ \texttt{;} \downarrow env[\boldsymbol{x} \mapsto \boldsymbol{n}], in', out} \qquad \textsc{print}\ \frac{env, in, \boldsymbol{e} \downarrow in', \boldsymbol{n}}{env, in, out, \texttt{print}\ \boldsymbol{e}\ \texttt{;} \downarrow env, in', out\ \boldsymbol{n}}$$

$\boxed{env, in, \boldsymbol{e} \downarrow in', \boldsymbol{n}}$ Evaluation rules for expressions $\boldsymbol{e}$.

$$\textsc{num}\ \frac{}{env, in, \boldsymbol{n} \downarrow in, \boldsymbol{n}} \qquad \textsc{input}\ \frac{}{env, \boldsymbol{n}\ in, \texttt{input} \downarrow in, \boldsymbol{n}} \qquad \textsc{plus}\ \frac{env, in_0, \boldsymbol{e}_1 \downarrow in_1, \boldsymbol{n}_1 \qquad env, in_1, \boldsymbol{e}_2 \downarrow in_2, \boldsymbol{n}_2}{env, in_0, (\ \boldsymbol{e}_1\ \texttt{+}\ \boldsymbol{e}_2\ ) \downarrow in_2, \boldsymbol{n}_1 + \boldsymbol{n}_2}$$

$$\textsc{var}\ \frac{\boldsymbol{x} \in env}{env, in, \boldsymbol{x} \downarrow in, env(\boldsymbol{x})}$$

Figure 1: Operational semantics for TINY.

These judgments for the TINY operational semantics can be understood somewhat like functions, where the elements to the left of $\downarrow$ are arguments and the elements to the right are results. The rules of each judgment can be understood somewhat like different cases in their judgment function. Let's jump straight into the individual judgments and rules, as this is the quickest way to understand the notation.

Programs. The judgment $env, in, out, \boldsymbol{p} \downarrow out'$ defines the complete evaluation of a program $\boldsymbol{p}$ under initial dynamic environment, , available inputs, *in*, and existing outputs *out*. Rule EMPTY indicates that the empty program (a program with no statements) evaluates to completion under any environment and inputs, producing the existing outputs. Since there are no premises, this rule is an *axiom*. Rule STATEMENT indicates that any nonempty program evaluates to completion under initial environment *env*, inputs *in*, and outputs *out*, if its first statement, $\boldsymbol{s}$, produces new environment env', remaining inputs in', and outputs out', when evaluated under the initial environment, inputs, and outputs *and* the rest of the program evaluates to completion under the environment, env', remaining inputs, in', and existing outputs, out', produced by evaluating the first statement.

Statements. The judgment $env, in, out, \boldsymbol{s} \downarrow env', in', out'$ defines the evaluation of a statement, $\boldsymbol{s}$. Notice that there is one rule for each kind of statement. Rule ASSIGN indicates that evaluating an assignment statement $\boldsymbol{x}$ `=` $\boldsymbol{e}$ `;` evaluates expression $\boldsymbol{e}$ under the initial environment environment and available inputs, using judgment $env, in, \boldsymbol{e} \downarrow in', \boldsymbol{n}$, and produces a new environment that extends the initial environment *env* by mapping $\boldsymbol{x}$ to $\boldsymbol{n}$, the result of evaluating $\boldsymbol{e}$. Whatever inputs, in', remained after evaluating $\boldsymbol{e}$ remain after evaluating $\boldsymbol{s}$. Assignment produces no new outputs. Rule PRINT is similar, but does not change the environment. Instead it produces one new output: the number $\boldsymbol{n}$, resulting from evaluating $\boldsymbol{e}$.

Expressions. The judgment $env, in, \boldsymbol{e} \downarrow in', \boldsymbol{n}$ defines the evaluation of an expression, $\boldsymbol{e}$. There is one rule for each kind of expression. Rule NUM indicates that a number, $\boldsymbol{n}$, evaluates to itself, without consuming inputs. Rule INPUT indicates that an `input` expression evaluates to the first remaining input, $\boldsymbol{n}$, and removes this input from the remaining inputs. Notice that this rule cannot be applied if there are *no* remaining inputs. Rule PLUS indicates that an addition expression evaluates the left expression, $\boldsymbol{e}_1$, (which may consume inputs), then the right expression, $\boldsymbol{e}_2$, (which may also consume inputs) and yields the arithmetic sum of the two results, with any inputs not consumed by evaluation of the subexpressions also remaining after evaluation of the addition expression. Rule VAR indicates that a use of variable $\boldsymbol{x}$ evaluates to the number to which $\boldsymbol{x}$ maps in the environment *env*. Notice that this rule applies only if such a mapping exists in *env*.