CSCI3180 Principles of Programming Languages

Ryan Chan

May 6, 2026

Abstract

This is a note for **CSCI3180 Principles of Programming Languages**.
Contents are adapted from the lecture notes of CSCI3180, prepared by Lauren Pick, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

1 Foundations **2**
- 1.1 Syntax and Semantics . . . 2
- 1.2 Formal Syntax Rules . . . 2
- 1.3 Abstract syntax trees (ASTs) . . . 5
- 1.4 Beyond BNF and Meta-variables . . . 6

2 Formal Proof Systems **9**
- 2.1 Proof Systems . . . 9
- 2.2 From Grammars to Proof System . . . 12
- 2.3 Predicate Logic . . . 12

3 Semantics Basics **14**
- 3.1 Introduction to Semantics . . . 14
- 3.2 SIMPL . . . 14
- 3.3 Small-step Operational Semantics . . . 15
- 3.4 Evaluation Order and Determinism . . . 17

Chapter 1

Foundations

1.1 Syntax and Semantics

Consider an arbitrary programming language $\mathcal{L}$ (a set of programs). We have:

- **Syntax**: describes which things are programs in $\mathcal{L}$

- **Semantics**: describes what happens when we run a program in $\mathcal{L}$

So, given the syntax for a programming language $\mathcal{L}$, with a set of values $\mathcal{V}$, we define

$$\mathsf{Interpreter}_{\mathcal{L}} : \mathcal{L} \to \mathcal{V}$$

Here the *interpreter* takes a program in $\mathcal{L}$ and runs it to produce a value in $\mathcal{V}$. For example,

$$\mathsf{Interpreter}_{\text{Python}}(\texttt{sorted([1, 4, 3, 2])}) = \texttt{[1, 2, 3, 4]}$$

This can be seen as one way of defining the semantics of the program.

A compiler translates from one language to another. For a program p in $\mathcal{L}_1$ and $\mathsf{Compiler} : \mathcal{L}_1 \to \mathcal{L}_2$ to be correct, we want

$$\mathsf{Interpreter}_{\mathcal{L}_1}(p) = \mathsf{Interpreter}_{\mathcal{L}_2}(\mathsf{Compiler}(p))$$

otherwise something is incorrect.

Notice that interpreters and compilers, in **theory**, are mathematical objects, but not in reality. They have implementations, and while we can treat implementations of programming languages as defining their semantics, it is helpful not to do this, since implementations of interpreters and compilers are themselves programs and can be buggy.

Furthermore, different implementations of the same programming language may exhibit *diverging behaviors*, especially in the presence of under-specification. We can see that it is helpful to know what programs are meant to do (i.e., their semantics) separately from their implementations.

1.2 Formal Syntax Rules

1.2.1 Formal Language

A formal language L over an alphabet Σ is a subset of Σ^*, which is the set of strings consisting of and making use of elements in Σ. The empty string is denoted by ε.

For example, a formal language L over $\Sigma = \{1, 2\}$ is a subset of Σ^*, e.g. $L = \{\varepsilon, 1, 2, 11, 22, 111, 222\}$.

One way to manipulate strings is using a rewriting system, which consists of rules that map strings in a formal language to other strings in the language. For a language L, we define a relation $\to$ over strings in L. For example, for $L = \{1, 2, 11\}$, we may have $\to = \{(1, 2), (2, 1), (1, 11), (11, 2)\}$. Then we can write $1 \to 2$, $2 \to 1$, $1 \to 11$, and $11 \to 2$.

We also have generative grammar, which is a kind of rewriting system over a language $(\Sigma \cup N)^*$, where

- the alphabet is partitioned into two sets: a set of non-terminal symbols N and a set of terminal symbols Σ,

- there is a distinguished start symbol in N,

- rewrite rules are called *production rules*; the left-hand side of each rule contains at least one non-terminal.

We can use generative grammars to define a language over the alphabet Σ. The language defined is the set of strings of terminal symbols that we can generate by starting with the start symbol S and following the production rules.

Example (Generative Grammar)**.** Consider $\Sigma \equiv \{a, b, c\}$.

Language ONE with only two strings $\{a, b\}$.

In this case, to generate a language over Σ, we have $N = \{S\}, \Sigma = \{a, b, c\}$ with grammar (production rules):

$$S \to a$$
$$S \to b$$

Language A-B where no a follows any b, e.g. $\varepsilon, ab, aab, acb$.

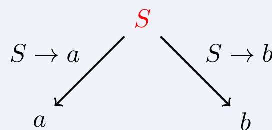
In this case, consider $N = \{S, A, B, C\}, \Sigma = \{a, b, c\}$ with production rules:

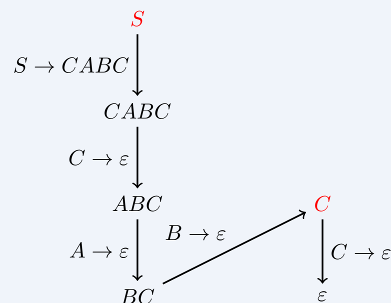
$$\begin{array}{ll} S \to CABC & \\ C \to \varepsilon & C \to cC \\ A \to \varepsilon & A \to aCA \\ B \to \varepsilon & B \to bCB \end{array}$$

Language MATCH where the number of as and bs is the same.

In this case, consider $N = \{S\}, \Sigma = \{a, b, c\}$ with production rules:

$$\begin{array}{ll} S \to \varepsilon & S \to c \\ S \to aSb & S \to bSa \\ S \to SS & \end{array}$$





1.2.2 Context-Free Grammar (CFGs)

The above grammars are context-free grammars, where the left-hand side of each production rule consists of a single non-terminal. It is often specified using Backus-Naur Form (BNF), where

- every production rule is written with $::=$ instead of $\to$,

- production rules with the same left-hand side are combined into a single rule, where different right-hand sides are separated with $|$, e.g. Language BOOL of boolean formulas with

- conjunction (and) $\wedge$
- disjunction (or) $\vee$
- negation (not) $\neg$

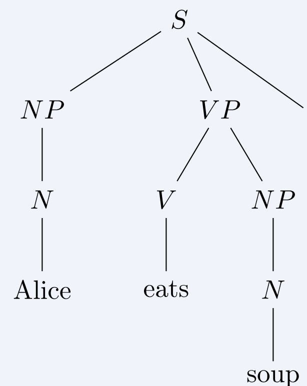
$$S ::= x \mid y \mid z \mid w \mid (S) \mid S \wedge S \mid S \vee S \mid \neg S$$

Example (BNF). Consider a BNF grammar for LUNCH^{-}.

$$\begin{aligned} S &::= NP\ VP. \\ NP &::= N \mid N\ PP \\ VP &::= V \mid V\ NP \mid V\ PP \mid V\ NP\ PP \\ V &::= \text{buys} \mid \text{eats} \\ N &::= \text{Alice} \mid \text{Bob} \mid \text{soup} \mid \text{salad} \mid \text{pizza} \\ PP &::= \text{in a bowl} \mid \text{in a teacup} \mid \text{in a house} \end{aligned}$$

Then we can derive a derivation tree (parse tree):

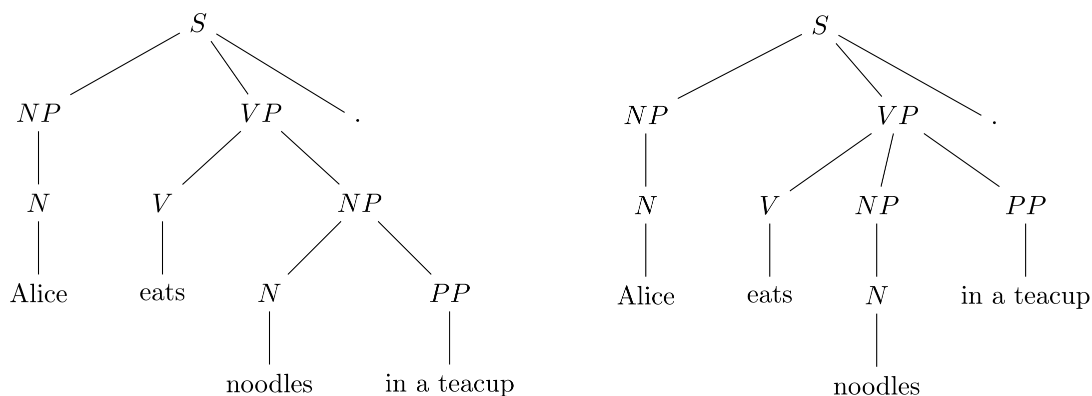
$$\begin{aligned} S &\to NP\ VP. \\ &\to N\ VP. \\ &\to \text{Alice}\ VP. \\ &\to \text{Alice}\ V NP. \\ &\to \text{Alice eats}\ NP. \\ &\to \text{Alice eats}\ N. \\ &\to \text{Alice eats soup.} \end{aligned}$$



Remark. A parse tree acts as a proof that a string is in a language (membership).

1.2.3 Ambiguity & *Dangling Else Problem*

A grammar is ambiguous if the same string has more than one parse tree, e.g.



Grammar serves an additional purpose, that is, it can be used to give structure to a string in a language. Semantics for the strings within a language are typically described with respect to this additional structure imposed on the string. Ambiguous grammar can thus lead to ambiguous meanings being assigned to strings.

For example, in the above examples, the left suggests that the noodles are in a teacup, whereas the right one suggests that Alice is located in a teacup. While one interpretation may be more likely than another and judgment can be based on semantics and background knowledge, the correct conclusion is not determined by the grammar alone.

This problem can manifest in the *dangling else problem* in a programming language context.

Consider language IF with the following grammar:

$$\begin{aligned} S &::= C \mid \text{ if } B \text{ then } S \mid \text{ if } B \text{ then } S \text{ else } S \\ B &::= \text{ true } \mid \text{ false } \mid \ldots \\ C &::= \ldots \end{aligned}$$

To deal with ambiguity, we can

1. change the grammar such that it is **not** ambiguous, i.e. same language with a different grammar.

For example, for BOOL:

$$S ::= x \mid y \mid z \mid w \mid (S) \mid S \wedge S \mid S \vee S \mid \neg S$$

we change to

$$\begin{aligned} S &::= D \vee S \mid D \\ D &::= C \wedge D \mid C \\ C &::= (S) \mid \neg C \mid x \mid y \mid z \mid w \end{aligned}$$

2. change the syntax, i.e. a different language with a different grammar.

For example, we change language IF to

$$S ::= C \mid \text{ if } B \text{ then } S \text{ end if } \mid \text{ if } B \text{ then } S \text{ else } S \text{ end if}$$

1.3 Abstract syntax trees (ASTs)

Before, we have $\mathsf{Interpreter}_{\mathcal{L}}$ for $\mathcal{L}$ that takes a program in $\mathcal{L}$ and produces a value. Typically, there is a frontend containing a *parser* and a backend containing an *evaluator*. The parser parses the input string according to a grammar and converts the input program into an intermediate representation called an abstract syntax tree (AST),

$$\mathsf{Parser}_{\mathcal{L}} : \mathcal{L} \to \mathrm{AST}(\mathcal{L})$$

and the evaluator then evaluates the AST to a value,

$$\mathsf{Evaluator}_{\mathrm{AST}(\mathcal{L})} : \mathrm{AST}(\mathcal{L}) \to \mathcal{V}$$

Then we have

$$\mathsf{Interpreter}_{\mathcal{L}} = \mathsf{Evaluator}_{\mathrm{AST}(\mathcal{L})} \circ \mathsf{Parser}_{\mathcal{L}}$$

The AST representation abstracts a parse tree and keeps only the relevant syntactic information from the parse tree, in particular, a subset of the production rules applied to obtain the parse tree.

To make an AST:

1. derive a parse tree annotated with (important) production rules applied,

2. remove terminal nodes,

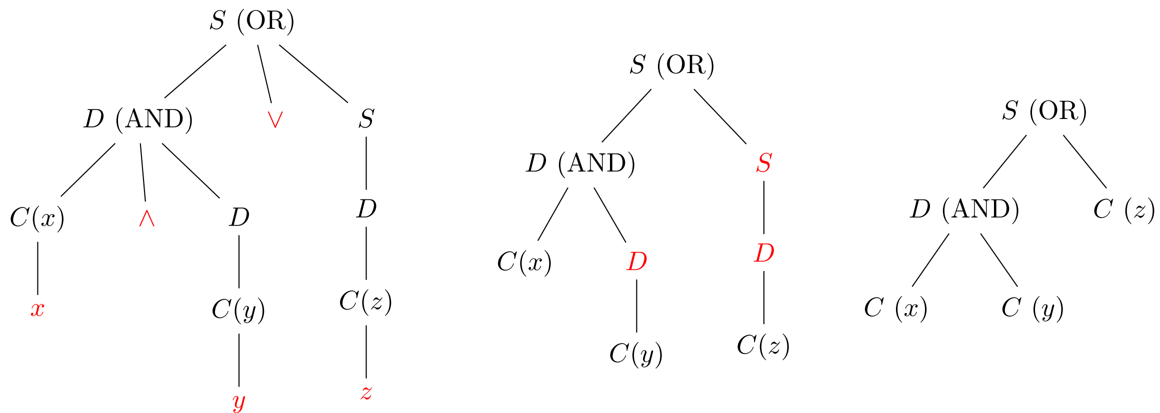
3. remove nodes expanded by production rules we do not care about; on removal, parent the node's children to its parent,

4. relabel non-terminal nodes with the annotated production rule names.

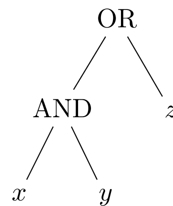
For example, for $x \wedge y \vee z$ with

$$\begin{aligned} S &::= D \vee S \mid D \\ D &::= C \wedge D \mid C \\ C &::= (S) \mid \neg C \mid x \mid y \mid z \mid w \end{aligned}$$

we have



finally we have



Notice that we have viewed ASTs as generic trees, but they are often trees with a specific structure, i.e. nodes with a certain label might always have a particular degree or children that have only certain kinds of labels. This structure can be captured by another grammar, where we introduce a production rule in the AST grammar for each production rule that we care about. For example, for the above BOOL, we might have the following grammar:

$$AST ::= OR(AST, AST) \mid AND(AST, AST) \mid NOT(AST) \mid x \mid y \mid z \mid w$$

This can be directly translated into an ADT, such as the following in OCaml:

```
type bool_ast = Or of bool_ast * bool_ast
  | And of bool_ast * bool_ast
  | Not of bool_ast
  | X
  | Y
  | Z
  | W
```

The `bool_ast` for $x \wedge y \vee z$ is `Or(And(X,Y),Z)`.

When implementing a parser, rather than generating a full parse tree and then an AST, we can generate the AST during parsing by applying the appropriate ADT constructor for a production rule.

1.4 Beyond BNF and Meta-variables

Many programming languages use formal grammars to specify syntax, yet they do not directly use BNF. Instead, they use **extensions** or **variants** of BNF.

For example, in the language BOOL, we have

$$Vars \cup \{\wedge, \vee, \neg, (,)\}$$

and in the grammar each variable in $Vars$ becomes a terminal in the production rules. To extend BOOL such that it allows more variables requires updating the BNF grammar with new production rules. However, when we want to allow infinitely many variable names, this becomes impossible as BNF grammars are finite.

We then use extensions of BNF which allow using additional notations outside of BNF, e.g. *meta-variables*. For example, let $BoolVars$ be an infinite set of variables and let v range over $BoolVars$; then v is a stand-in for any element of $BoolVars$. Then we can still construct parse trees using BNF-like grammars and derive ASTs for them as before.

For example, consider the BNF grammar:

$$\begin{aligned} S &::= Z \mid Z + S \\ Z &::= -N \mid N \\ N &::= DN \mid D \\ D &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

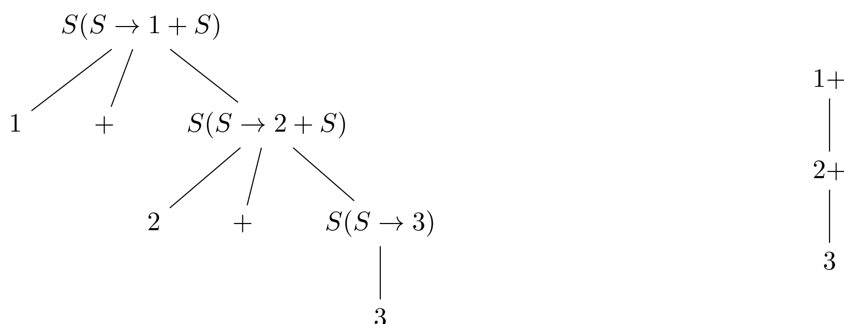
We typically write this as:

For z ranging over integers:

$$S ::= z \mid z + S$$

Here z is an integer, which is a semantic constraint, and this form of grammar is technically an extension of BNF.

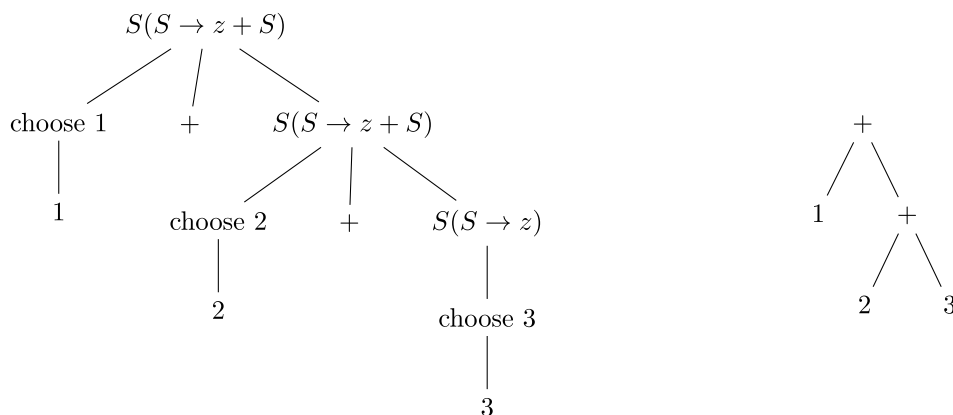
For example, we have a parse tree and AST for $1 + 2 + 3$ as:



However, consider tracking chosen z values, treating z as if generated by an infinite set of production rules:

$$z \in Z ::= \cdots \mid -1 \mid 0 \mid 1 \mid \ldots$$

Then we have

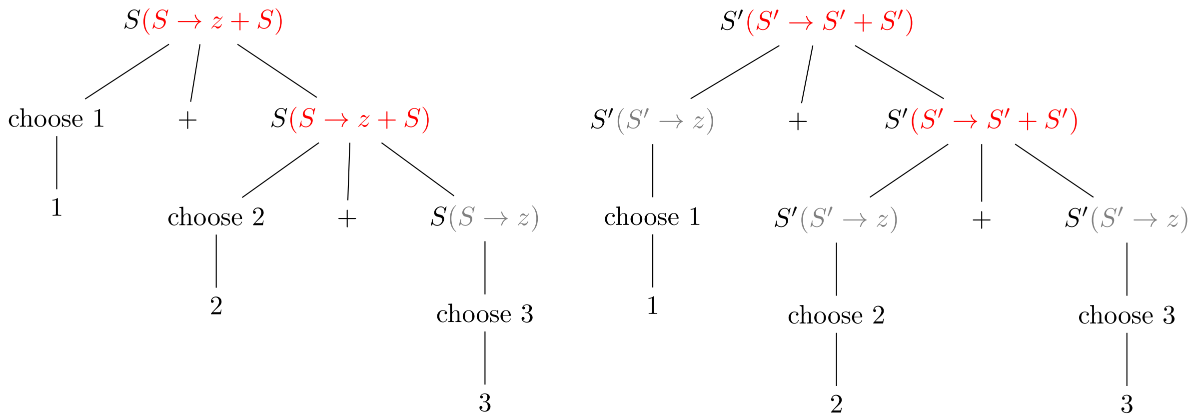


Note that an AST is an abstraction of the original parse tree that retains semantically meaningful parts. Choices of meta-variables are usually important.

For the above example, there is another grammar that generates the same language, i.e.

$$S' ::= z \mid S' + S'$$

For example, we have



which give the same AST.

So consider $S' ::= z \mid S' + S'$. If $+$ is left-associative, then the left-most $+$ will bind most tightly. We then expect this $+$ to be closest to the leaves in the AST, and the left operation is evaluated first.

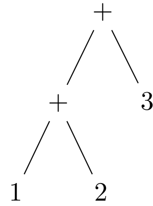


Figure 1.1: Left Associative

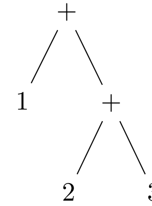


Figure 1.2: Right Associative

To further reduce ambiguity, we can add a meta-variable a that ranges over strings generated by the grammar $S' ::= z \mid S' + S'$, i.e.

$$a \in S' ::= z \mid a_1 + a_2$$

where $+$ is right-associative (by declaring the type of associativity to reduce ambiguity).

Then consider BOOL again. We can use meta-variables and BNF-like syntax to allow variable names to be anything in the possibly infinite set $BoolVars$, e.g.

$$S ::= v \mid (S) \mid S \land S \mid S \lor S \mid \neg S$$

and to remove ambiguity, we can specify the *precedence* and *associativity* of $\land, \lor, \neg$.

The OCaml ADT for the structure of the ASTs for BOOL^+ could be something like

```
type bool_ext_ast = Or of bool_ext_ast * bool_ext_ast
  | And of bool_ext_ast * bool_ext_ast
  | Not of bool_ext_ast
  | Var of string
```

Chapter 2

Formal Proof Systems

In the previous chapter we mentioned that we can prove the membership of a string in a language by giving a parse tree. For example, for a language Nat with start symbol N and terminals $z, s,$ (,), we have

$$N ::= z \mid s(N)$$

and to prove $s(s(z)) \in \mathbf{Nat}$, we have

$$\begin{aligned} N &\to s(N) \\ &\to s(s(N)) \\ &\to s(s(z)) \end{aligned}$$

Then, how do we perform reasoning in a completely syntactic way rather than semantic? We use a proof system.

2.1 Proof Systems

Proof systems give us a way of performing reasoning in a completely syntactic way. A proof system has three components:

1. a formal language L of statements,

2. a set of inference rules (aka proof rules) that show how to prove statements. These typically have the following form, where the *premises* have zero or more statements and the *conclusion* is a single statement:

$$\frac{\textit{premises}}{\textit{conclusion}}$$

3. a subset of the inference rules called *axioms*, whose sets of premises are empty — we may omit the horizontal line when writing such rules.

2.1.1 Propositional Logic

Statements of propositional logic range over a language consisting of propositional variables and connectives, where propositional variables represent propositions, which may either be true ($\top$) or false ($\bot$).

Consider the grammar for **propositional logic**:

$$\begin{aligned} P &::= A \mid P \overbrace{\circledcirc}^{\text{binary}} P \mid \overbrace{\neg}^{\text{not}} P \mid \overbrace{\top}^{\text{true}} \mid \overbrace{\bot}^{\text{false}} \\ \circledcirc &::= \underbrace{\wedge}_{\text{and}} \mid \underbrace{\vee}_{\text{or}} \mid \underbrace{\Rightarrow}_{\text{implies}} \end{aligned}$$

Capital letters like A, B, C range over a set of propositional variables. For example, we can have $A \Rightarrow A$, $A \wedge B \Rightarrow A$, $A \wedge B \vee C$, $A \wedge B \Rightarrow A \vee B$.

While the semantics of a programming language is defined in terms of its abstract syntax (like ASTs) rather than concrete syntax, we typically write the concrete syntax in place of the abstract syntax and indicate the structure in the AST using parentheses when it is ambiguous. For example, we write $(A \lor B) \land C$ rather than `AND(OR(A, B), C)`.

To remove ambiguity, we assume:

1. $\land$ has precedence over $\lor$ and $\Rightarrow$, and $\lor$ has precedence over $\Rightarrow$,
2. $\neg$ has precedence over all other connectives,
3. all connectives other than $\Rightarrow$ are left-associative,
4. $\Rightarrow$ is right-associative.

Note that "has precedence over" means "binds more tightly", e.g. $A \land B \lor C$ is interpreted as $(A \land B) \lor C$.

As for associativity, for left-associative operators, the further to the left an operator is, the more tightly it binds, e.g. $A \land B \land C \land D$ is treated as $((A \land B) \land C) \land D$.

2.1.2 Natural Deduction System

Again we consider propositional logic:

$$\begin{aligned} S &::= [P] \mid P \\ P &::= A \mid P \circledcirc P \mid \neg P \mid \top \mid \bot \\ \circledcirc &::= \land \mid \lor \mid \Rightarrow \end{aligned}$$

where P is a propositional statement, and $[P]$ is an assumption. Therefore, plain P is the conclusion, and $[P]$ is an assumption during the proof.

Here let P, Q, R be meta-variables ranging over arbitrary propositional formulas. For each logical connective, there are *introduction* rules (which introduce the connective in the conclusion) and *elimination* rules (which eliminate the connective from one of the premises in the conclusion). Then we have:

- Truth and Falsity

$$(\text{UNIT})\ \frac{}{\top} \qquad\qquad (\text{EX FALSO})\frac{\bot}{P}$$

For introduction, $\top$ is always true. For elimination, if you have $\bot$ (false / contradiction), you can conclude any proposition P (provable but not necessarily true).

- Negation

$$(\neg\ \text{INTRO})\ \frac{\begin{array}{c}[P]\\ \vdots \\ \hline \bot\end{array}}{\neg P} \qquad\qquad (\neg\ \text{ELIM})\ \frac{P \quad \neg P}{\bot}$$

For introduction, if assuming P leads to a contradiction, then you can conclude $\neg P$. For elimination, if both P and $\neg P$ exist, you get a contradiction.

- Conjunction

$$(\land\ \text{INTRO})\ \frac{P \quad Q}{P \land Q} \qquad (\land\ \text{ELIM 1})\ \frac{P \land Q}{P} \qquad (\land\ \text{ELIM 2})\ \frac{P \land Q}{Q}$$

- Disjunction

$$(\lor\ \text{INTRO 1})\ \frac{P}{P \lor Q} \qquad (\lor\ \text{INTRO 2})\ \frac{Q}{P \lor Q} \qquad (\lor\ \text{ELIM})\ \frac{P \lor Q \quad \begin{array}{c}[P]\\ \vdots \\ \hline R\end{array} \quad \begin{array}{c}[Q]\\ \vdots \\ \hline R\end{array}}{R}$$

- **Implication**

$$(\Rightarrow \text{INTRO})\ \frac{\begin{matrix}[P]\\ \vdots \\ \hline Q\end{matrix}}{P \Rightarrow Q} \qquad (\text{MODUS PONENS})\ \frac{P \quad P \Rightarrow Q}{Q}$$

- **Assumption**

$$\frac{\begin{matrix}[P]\\ \vdots\end{matrix}}{P}$$

Then we can formalize an informal proof:

Example. Consider

If Alice likes tea and Alice likes bread, then Alice likes tea.

So to prove $T \wedge B \Rightarrow T$, we consider

$$(\Rightarrow \text{INTRO})\ \frac{\begin{matrix}\underline{[T \wedge B]}\\ \boxed{?} \\ \hline T\end{matrix}}{T \wedge B \Rightarrow T}$$

using assumption, we have

$$(\text{ASSUMPTION})\ \frac{\begin{matrix}\underline{[T \wedge B]}\\ \underline{T \wedge B}\\ \boxed{?} \\ \hline T\end{matrix}}{T \wedge B \Rightarrow T}$$

using $\wedge$ELIM 1, we have

$$(\wedge\text{ELIM 1})\ \frac{\begin{matrix}\underline{[T \wedge B]}\\ \underline{T \wedge B}\\ T\end{matrix}}{T \wedge B \Rightarrow T}$$

This is a derivation tree (proof tree).

We can modify the form of statements to keep track of assumptions in a context Γ, which is a comma-separated list of propositional logic statements P that we treat as assumptions, i.e.

$$S ::= \Gamma \vdash P$$
$$\Gamma ::= \Gamma, P \mid \varnothing$$

Here Γ ranges over possible contexts (lists of assumptions). Then we have

$$(\text{ASSUMPTION})\ \frac{}{\Gamma \vdash P} \text{ if } P \text{ is in } \Gamma$$

$$(\text{UNIT})\ \frac{}{\Gamma \vdash \top} \qquad (\text{EX FALSO})\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \qquad (\neg\ \text{INTRO})\ \frac{\Gamma, P \vdash \bot}{\Gamma \vdash \neg P} \qquad (\neg\ \text{ELIM})\ \frac{\Gamma \vdash P \quad \Gamma \vdash \neg P}{\Gamma \vdash \bot}$$

$$(\wedge\ \text{INTRO})\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \qquad (\wedge\ \text{ELIM 1})\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad (\wedge\ \text{ELIM 2})\ \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}$$

$$(\Rightarrow \text{INTRO})\ \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \qquad (\text{MODUS PONENS})\ \frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}$$

$$(\vee\ \text{INTRO 1})\ \frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad (\vee\ \text{INTRO 2})\ \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

$$(\vee\ \text{ELIM})\ \frac{\Gamma \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R}$$

Note that the typical notation omits $\varnothing$ when writing Γ. For example, instead of writing $\varnothing \vdash \top$, we write $\vdash \top$.

2.2 From Grammars to Proof System

Recall language Nat with start symbol N and terminals z, s:

$$N ::= z \mid s(N)$$

where we have an equivalent set of inference rules:

$$\text{ZERO}\ \frac{-}{z} \qquad\qquad \text{SUCC}\ \frac{N}{s(N)}$$

So to prove $s(s(z)) \in \text{Nat}$, we consider

$$\begin{aligned} N &\to s(N) \\ &\to s(s(N)) \\ &\to s(s(z)) \end{aligned}$$

or

$$\cfrac{\cfrac{\cfrac{-}{z}\,(\text{ZERO})}{s(z)}\,(\text{SUCC})}{s(s(z))}\,(\text{SUCC})$$

We can also convert production rules to inference rules with notation $e : E$, i.e. e can be generated from E. Again recall the language BOOL:

$$\begin{aligned} S &::= D \lor S \mid D \\ D &::= C \land D \mid C \\ C &::= (S) \mid \neg C \mid x \mid y \mid z \mid w \end{aligned}$$

We have

$$(\text{OR})\ \frac{d : D \quad s : S}{d \lor s : S} \quad \frac{d : D}{d : S} \qquad\qquad (\text{AND})\ \frac{c : C \quad d : D}{c \land d : D} \quad \frac{c : C}{c : D}$$

$$\frac{s : S}{(s) : C} \quad (x)\ \frac{}{x : C} \quad (y)\ \frac{}{y : C} \quad (z)\ \frac{}{z : C} \quad (w)\ \frac{}{w : C}$$

Then we can:

1. prove that the string can be generated by S using the proof rules,

2. construct a tree by making a node for each important derivation step, where a node's first named descendant in the tree is its child.

For example, we can have:

$$\cfrac{\cfrac{\cfrac{}{x : C}\,(x)}{x : D} \quad \cfrac{\cfrac{\cfrac{}{y : C}\,(y)}{y : D}}{y : S}}{x \lor y : S}\,(\text{OR})$$

2.3 Predicate Logic

Instead of propositional variables, we have:

- predicate symbols, which are applied to zero or more terms,

- function symbols, which are applied to zero or more terms,

- quantifiers $\forall, \exists$, which are applied to variables.

A term is either a variable or an n-ary function symbol applied to n terms, e.g.

$$\forall x.\ 1 + x < 0 \quad \text{and} \quad \forall x.\ \forall y.\ \forall z.\ x > y \land y > z \Rightarrow x > z$$

Then for

If Alice likes tea and Alice likes bread, then Alice likes tea.

we have

$$Likes(Alice, tea) \wedge Likes(Alice, bread) \Rightarrow Likes(Alice, tea)$$

For

If Alice likes tea and tea is in the bag, then Alice likes something in the bag.

we have

$$Likes(Alice, tea) \wedge InBag(tea) \Rightarrow \exists thing.\ Likes(Alice, thing) \wedge InBag(thing)$$

Example (AST and Proof Rules). Consider Language ADDITION. We add a meta-variable a that ranges over strings generated by the grammar $S ::= z \mid S' + S'$:

$$a \in S' ::= z \mid a_1 + a_2 \qquad \text{where } + \text{ is right-associative}$$

We then try to define an "evaluation step" with a proof system whose statements (judgments) look like

$$a_1 \rightarrow a_2$$

We have the inference rules:

$$\text{(PLUS)}\ z_1 + z_2 \rightarrow z_3 \quad \text{if} \quad z_3 = z_1 + z_2$$

$$\text{(LEFT)}\ \frac{a_1 \rightarrow a_1'}{a_1 + a_2 \rightarrow a_1' + a_2}$$

$$\text{(RIGHT)}\ \frac{a \rightarrow a'}{z + a \rightarrow z + a'}$$

The program will then evaluate to an integer z. Now consider evaluating $1 + 2 + 3$.

For PLUS, we have $z_1 + z_2$; however, we cannot instantiate meta-variables in $z_1 + z_2$ to get $1+2+3$, since both z_1 and z_2 must be integers (single-node ASTs).

However, for LEFT and RIGHT, we can match $1 + 2 + 3$. For LEFT, we have $a_1 + a_2$, where a_i ranges over any expression in the language, so any sub-AST can replace a_i. This is the same for RIGHT.

Consider LEFT. After instantiating $a_1 = 1$ and $a_2 = 2 + 3$, we have

$$\frac{1 \rightarrow a_1'}{1 + 2 + 3 \rightarrow a_1' + 2 + 3}$$

To finish this proof, we need to prove the premise $1 \rightarrow a_1'$ for some instantiation of a_1'. However, no rule applies to 1, since it is already a value, so the proof fails.

Next, consider RIGHT:

$$\frac{2 + 3 \rightarrow a'}{1 + 2 + 3 \rightarrow 1 + a'}$$

To finish the proof, we need to prove $2 + 3 \rightarrow a'$. This does not fail; instead, we must continue evaluating $2 + 3$.

Now we can apply PLUS by instantiating $z_1 = 2$ and $z_2 = 3$:

$$2 + 3 \rightarrow z_3 \quad \text{if} \quad 2 + 3 = z_3$$

From the side condition, $z_3 = 5$, so $a' = 5$. Hence,

$$\frac{2 + 3 \rightarrow 5}{1 + 2 + 3 \rightarrow 1 + 5}$$

We then evaluate $1 + 5$, which is similar to $2 + 3$. By applying PLUS, we obtain:

$$1 + 5 \rightarrow 6$$

Chapter 3

Semantics Basics

3.1 Introduction to Semantics

Consider the following C++14 program:

```
int main (int argc, char *argv[]) {
  int i = 1;
  int v = 2;
  i = i++ + v;
  printf("%d\n", i);

  return 0;
}
```

Using the `clang` compiler it prints 3, while using the `VS` compiler, it prints 4. However, what should it do and what is it supposed to do? The answer comes from the semantics of programming languages.

Most programming languages have semantics described in natural language, which is preferable for many purposes but can be *imprecise* and *ambiguous*. For example, consider a programming language with a built-in function `sort`, where the documentation says:

The `sort` function takes an array and a comparator function as arguments. It sorts the array using the comparator function and returns it.

This description is *imprecise*, as there is no information on whether it modifies the original array or creates a new one, nor even the order of sorting.

We can modify the description to be more precise and less ambiguous, yet natural language itself is hard to make fully precise. The ambiguity and imprecision of natural language can lead to diverging implementations of the same programming language, which leads to unexpected behavior.

Therefore, we want formal semantics that give meaning to programs in a **precise**, **unambiguous**, and **concise** way.

Different ways include:

- **Operational semantics**: how is a program executed on an abstract machine?

- **Denotational semantics**: what does the program mean in terms of mathematical objects?

- **Axiomatic semantics**: what should we be able to prove about this program?

3.2 SIMPL

SIMPL is the Simple IMPerative Language. This is an *imperative* programming language, where programs are expressed in terms of sequences of commands or statements. It has common imperative control structures, such as while loops and if-statements. It also has *locations* that hold integer or boolean values,

which can be *dereferenced* using the ! operator.

The grammar of SIMPL is as below:

$$
\begin{array}{rll}
e \in \langle expr \rangle ::= & b \mid z \mid \textbf{skip} & \\
& \mid r := e & \text{assignment} \\
& \mid !r & \text{dereference} \\
& \mid e_1; e_2 & \text{sequencing} \\
& \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \text{if statement} \\
& \mid \textbf{while } e_1 \textbf{ do } e_2 & \text{while loop} \\
& \mid e_1 \odot e_2 & \\
\odot \in \langle op \rangle ::= & + \mid - \mid * \mid / \mid \leq &
\end{array}
$$

Here r ranges over a set of allowed location names, b ranges over the set of Booleans $\mathbb{B}$, and z ranges over the set of integers $\mathbb{Z}$.

We remove ambiguity by making the following assumptions:

1. $*$ and $/$ have precedence over $+$ and $-$, so that $e_1 + e_2 \diamond e_3$ is interpreted as $e_1 + (e_2 \diamond e_3)$, where $\diamond \in \{*, /\}$,
2. operators $+, -, *, /$ all bind more tightly than $\leq$,
3. the sequencing operator ; is right-associative and binds the least tightly,
4. operators $\odot$ are left-associative.

For SIMPL, we define the set of values that the program evaluates to by

$$v \in \mathit{Values} ::= z \mid b \mid \textbf{skip}$$

Example.

Program 1:

```
1 x := 3;
2 y := 1;
3 while 2 < !x do (y := !x * !y; x := !x - 1)
```

Program 2:

```
1 x := 3;
2 z := (y := 10; if !y ≤ !x then !x else !y);
3 !z
```

Program 3:

```
1 x := 1;
2 while 0 ≤ !x do x := !x + 1
```

Program 4:

```
1 x := 1;
2 while !x do x := !x - 1
```

3.3 Small-step Operational Semantics

Since programs describe computations, we can think of the meaning of a program as the computation it describes. This is essentially *operational semantics*, where the meaning of programs is defined by describing how a program should be executed on an abstract machine.

This definition is given via a proof system in which rules' statements (judgments) describe the execution steps that the abstract machine can perform. If we can prove the steps can be taken, we can take them.

In small-step operational semantics, judgments describe a *single* atomic computation step that the abstract machine can perform. Each step is called a *reduction*. Evaluating a program is then performing reductions until a value is reached.

In SIMPL, the judgments in particular will involve reducing *configurations*, which are pairs of a program and a program *state* that keeps track of the context in which the program should be executed. For SIMPL we use a *store*, which is a map from location names to the values stored in them. For example:

$$\{\texttt{var1} \mapsto 1, \texttt{var2} \mapsto 2\}$$

where locations `var1` and `var2` store values 1 and 2 respectively.

This store is a *mutable store* in which the values may change.

The operational semantics rules for SIMPL are as follows:

$$\text{(ASSIGN-BOOL)}\ (r := b, s) \to (\textbf{skip}, s[r \mapsto b]) \qquad \text{(DEREF-BOOL)}\ (!r, s) \to (b, s) \text{ if } s[r] = b$$

$$\text{(ASSIGN-INT)}\ (r := z, s) \to (\textbf{skip}, s[r \mapsto z]) \qquad \text{(DEREF-INT)}\ (!r, s) \to (z, s) \text{ if } s[r] = z$$

$$\text{(ASSIGN-EXPR)}\ \frac{(e, s) \to (e', s')}{(r := e, s) \to (r := e', s')}$$

$$\text{(SEQ-1)}\ (\textbf{skip}; e, s) \to (e, s) \qquad \text{(SEQ-2)}\ \frac{(e_1, s) \to (e_1', s')}{(e_1; e_2, s) \to (e_1'; e_2, s')}$$

$$\text{(IF-TRUE)}\ (\textbf{if}\ \text{true}\ \textbf{then}\ e_2\ \textbf{else}\ e_3, s) \to (e_2, s)$$

$$\text{(IF-FALSE)}\ (\textbf{if}\ \text{false}\ \textbf{then}\ e_2\ \textbf{else}\ e_3, s) \to (e_3, s)$$

$$\text{(IF-EXPR)}\ \frac{(e_1, s) \to (e_1', s')}{(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, s) \to (\textbf{if}\ e_1'\ \textbf{then}\ e_2\ \textbf{else}\ e_3, s')}$$

$$\text{(WHILE)}\ (\textbf{while}\ e_1\ \textbf{do}\ e_2, s) \to (\textbf{if}\ e_1\ \textbf{then}\ (e_2; \textbf{while}\ e_1\ \textbf{do}\ e_2)\ \textbf{else}\ \textbf{skip}, s)$$

$$\text{(OP-1)}\ \frac{(e_1, s) \to (e_1', s')}{(e_1 \odot e_2, s) \to (e_1' \odot e_2, s')} \qquad \text{(OP-2)}\ \frac{(e_2, s) \to (e_2', s')}{(v \odot e_2, s) \to (v \odot e_2', s')}$$

$$\text{(LT)}\ (z_1 \leq z_2, s) \to (b, s)\ \underbrace{\text{if } b = (z_1 \leq z_2)}_{\text{side condition}}$$

Note that there are some side conditions that must hold for some rules above to be applicable.

Then we can use these rules to do evaluation. Considering Program 1:

```
x := 3;
y := 1;
while 2 < !x do (y := !x * !y; x := !x - 1)
```

Here we use Δ for `y := !x * !y; x := !x - 1` and $\mathbf{W}$ for `while 2 <= !x do` Δ. The initial configuration has the form $(e_1; e_2, \{\})$, where e_1 is `x := 3` and e_2 is `y := 1;` $\mathbf{W}$.

In order to take a step to the next configuration, we need to make sure that the step is derivable using the operational semantics. We can reason about the structure of the program and the rules in the semantics to see if a step can be taken.

So in order to take a step from an expression of this form, we need a rule whose left-hand side has the form $(e_1; e_2, s)$, where e_1 is not **skip**:

$$\text{(SEQ-2)}\ \frac{(x := 3, \{\}) \to (e_1', s')}{(x := 3; y := 1; \mathbf{W}, \{\}) \to (e_1'; e_2, s')}$$

To derive this we need to derive its premise. We choose to apply ASSIGN-INT:

$$\text{(ASSIGN-INT)}\ (x := 3, \{\}) \to (\textbf{skip}, \{x \mapsto 3\})$$

For ASSIGN-EXPR, we would need to take a step from $(3, s)$, but there is no rule for this judgment. Thus we have the full derivation tree:

$$\dfrac{\dfrac{}{(x := 3, \{\}) \to (\textbf{skip}, \{x \mapsto 3\})}\ \text{(ASSIGN-INT)}}{(x := 3; y := 1; \mathbf{W}, \{\}) \to (\textbf{skip}; y := 1; \mathbf{W}, \{x \mapsto 3\})}\ \text{(SEQ-2)}$$

We have the full evaluation as follow:

$$\begin{aligned}
(x := 3; y := 1; \mathbf{W}, \{\}) &\to (\textbf{skip}; y := 1; \mathbf{W} &&, \{x \mapsto 3\}) \\
&\to (y := 1; W &&, \{x \mapsto 3\}) \\
&\to (\textbf{skip}; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (\mathbf{W} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (\textbf{if } 2 <=!x \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (\textbf{if } 2 <= 3 \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (\textbf{if } \text{true} \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (\Delta; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (y := 3*!y; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (y := 3 * 1; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (y := 3; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 1\}) \\
&\to (\textbf{skip}; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 3\}) \\
&\to (x :=!x - 1; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 3\}) \\
&\to (x := 3 - 1; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 3\}) \\
&\to (x := 2; \mathbf{W} &&, \{x \mapsto 3, y \mapsto 3\}) \\
&\to (\textbf{skip}; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (\mathbf{W} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (\textbf{if } 2 <=!x \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (\textbf{if } 2 <= 2 \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (\textbf{if } \text{true} \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (\Delta; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (y := 2*!y; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (y := 2 * 3; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (y := 6; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 3\}) \\
&\to (\textbf{skip}; x :=!x - 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 6\}) \\
&\to (x :=!x - 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 6\}) \\
&\to (x := 2 - 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 6\}) \\
&\to (x := 1; \mathbf{W} &&, \{x \mapsto 2, y \mapsto 6\}) \\
&\to (\textbf{skip}; \mathbf{W} &&, \{x \mapsto 1, y \mapsto 6\}) \\
&\to (\mathbf{W} &&, \{x \mapsto 1, y \mapsto 6\}) \\
&\to (\textbf{if } 2 <=!x \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 1, y \mapsto 6\}) \\
&\to (\textbf{if } 2 <= 1 \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 1, y \mapsto 6\}) \\
&\to (\textbf{if } \text{false} \textbf{ then } (\Delta; \mathbf{W}) \textbf{ else skip} &&, \{x \mapsto 1, y \mapsto 6\}) \\
&\to (\textbf{skip} &&, \{x \mapsto 1, y \mapsto 6\})
\end{aligned}$$

3.4 Evaluation Order and Determinism

3.4.1 Determinism

The semantics we looked at for SIMPL are *deterministic*, meaning each program has only one possible behavior. For small-step operational semantics, this means that for every expression that can be reduced by the semantics, there is exactly one way to reduce it.

We also made choices like evaluation order to obtain deterministic semantics. For example, for the following left-to-right evaluation order:

$$\text{(OP-1)}\ \frac{(e_1, s) \to (e_1', s')}{(e_1 \odot e_2, s) \to (e_1' \odot e_2, s')} \qquad \text{(OP-2)}\ \frac{(e_2, s) \to (e_2', s')}{(v \odot e_2, s) \to (v \odot e_2', s')}$$

We can instead use a right-to-left evaluation order:

$$\text{(OP-1)}\ \frac{(e_2, s) \to (e_2', s')}{(e_1 \odot e_2, s) \to (e_1 \odot e_2', s')} \qquad \text{(OP-2)}\ \frac{(e_1, s) \to (e_1', s')}{(e_1 \odot v, s) \to (e_1' \odot v, s')}$$

Yet these yield the same results for all SIMPL programs that do not interact with the store.

Such programs have no *side effects*, and functions like these are called *pure* functions that have no side effects and have deterministic results. They can be regarded as implementations of mathematical functions.

> **Remark.** Side effects include any read/write to any kind of state.

However, most general-purpose programming languages have side effects, which include things like updating memory and writing to files, leading to different results from different evaluation orders. In the context of SIMPL, the only side effects are store updates.

Most languages are implemented with a left-to-right evaluation order. Many also implement something called *short-circuiting* for boolean operators, where if the boolean value of the expression can be determined from evaluating only the left operand, the right one will not be evaluated.

3.4.2 Nondeterminism

We can include both left-to-right and right-to-left rules in the small-step operational semantics. Then we have a nondeterministic semantics, which may lead programs to have different observable behaviors. If we allow nondeterministic semantics, then a single program could potentially be evaluated to several different values. Then the interpreter should be a relation:

$$\mathsf{Interpreter}_L \subseteq L \times V$$

where for a program e and value v:

$$(e, v) \in \mathsf{Interpreter}_L \Leftrightarrow e \text{ can evaluate to } v \text{ according to the semantics.}$$

There are cases where nondeterministic semantics are desirable. For example, the generation of random numbers. Consider adding the ability to generate random integers in SIMPL via the expression **randInt**. We then have:

$$(\mathbf{randInt}, s) \to (z, s)$$

Note that this does not enforce randomness, i.e. the value generated by **randInt** could always be the integer 1 or a pseudorandom number.

Another case where nondeterministic semantics might be accepted is for *concurrency*, where programs are made up of parallel compositions of different programs that run at the same time. This can speed up overall computation, yet leads to nondeterministic semantics. For example, we have:

$$(x := 1 \parallel y := 2); !y$$

It is common to model the semantics of concurrent programs using *interleaving semantics*, in which the semantics of a parallel composition of programs is defined to be the interleaving of the semantic steps of the composed programs. For example:

$$\text{(LEFT-STEP)}\ \frac{(e_1, s) \to (e_1', s')}{(e_1 \parallel e_2, s) \to (e_1' \parallel e_2, s')} \qquad \text{(RIGHT-STEP)}\ \frac{(e_2, s) \to (e_2', s')}{(e_1 \parallel e_2, s) \to (e_1 \parallel e_2', s')}$$

$$\text{(LEFT-SKIP)}(\mathbf{skip} \parallel e, s) \to (e, s) \qquad \text{(RIGHT-SKIP)}(e \parallel \mathbf{skip}, s) \to (e, s)$$

Then the four evaluations for $(x \coloneqq 1 \parallel y \coloneqq 2); !y$, starting from the empty store $\{\}$, allowed by the semantics are as follows:

Evaluation 1:

$$\begin{aligned}
((x \coloneqq 1 \parallel y \coloneqq 2); !y, \{\}) &\to ((\mathbf{skip} \parallel y \coloneqq 2); !y, && \{x \mapsto 1\}) \\
&\to (y \coloneqq 2; !y, && \{x \mapsto 1\}) \\
&\to (\mathbf{skip}; !y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (!y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (2, && \{x \mapsto 1, y \mapsto 2\})
\end{aligned}$$

Evaluation 2:

$$\begin{aligned}
((x \coloneqq 1 \parallel y \coloneqq 2); !y, \{\}) &\to ((\mathbf{skip} \parallel y \coloneqq 2); !y, && \{x \mapsto 1\}) \\
&\to ((\mathbf{skip} \parallel \mathbf{skip}); !y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (\mathbf{skip}; !y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (!y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (2, && \{x \mapsto 1, y \mapsto 2\})
\end{aligned}$$

Evaluation 3:

$$\begin{aligned}
((x \coloneqq 1 \parallel y \coloneqq 2); !y, \{\}) &\to ((x \coloneqq 1 \parallel \mathbf{skip}); !y, && \{y \mapsto 2\}) \\
&\to (x \coloneqq 1; !y, && \{y \mapsto 2\}) \\
&\to (\mathbf{skip}; !y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (!y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (2, && \{x \mapsto 1, y \mapsto 2\})
\end{aligned}$$

Evaluation 4:

$$\begin{aligned}
((x \coloneqq 1 \parallel y \coloneqq 2); !y, \{\}) &\to ((x \coloneqq 1 \parallel \mathbf{skip}); !y, && \{y \mapsto 2\}) \\
&\to ((\mathbf{skip} \parallel \mathbf{skip}); !y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (\mathbf{skip}; !y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (!y, && \{x \mapsto 1, y \mapsto 2\}) \\
&\to (2, && \{x \mapsto 1, y \mapsto 2\})
\end{aligned}$$

Note that while the semantics is nondeterministic, the result of the program is deterministic regardless of how the program is evaluated. However, this is not always the case for all programs.