Welcome

*Programming Language Foundations*

The foundations of programming languages are introduced at an advanced level. Course topics include but are not limited to the foundational and conceptual subtleties in programming language design and implementation, the functional programming paradigm, the logic programming paradigm, parallel programming languages, formal specification of syntax and semantics, and techniques for (semi-)automatic correctness analysis of programs.

- **Instructor:** Robert van Engelen

- **Syllabus:**
  
  http://www.cs.fsu.edu/~engelen/courses/COT5315

- **Prerequisites:** COP4020 and MAD3105, or equivalent
Introduction  What is it all about?

Programming Language Foundations introduces a formal description of programming languages. The course teaches the principles and semantics of programming languages.

Among the topics covered are:

- A brief history of programming languages (Ch. 1)
- Grammars, parsing, and Post systems (Ch. 2 and Section 4.5)
- Lambda calculus, term rewriting, and functional programming (Ch. 7)
- Denotational semantics (Ch. 8)
- Axiomatic semantics (Ch. 9)
- Programming in Prolog (Ch. 6)

The other chapters of the textbook are not discussed.
Chapter 1  History of Programming Languages

Chapter 1 describes the development of (formal) languages and is excellent reading material.

We will discuss the Chapter’s main topics.

The evolution of the alphabet as a system of writing by humans has gone through several stages:

- Writing composed of individual *pictographs*, pictures of concepts (4000–3000BC)
- Stylized *ideographs* replaced pictographs as symbols for abstract ideas, their meaning needs to be communicated (3000BC)
- If the underlying meaning of an ideograph depends on the sounds of the spoken language, the symbol is a *phonogram*
- A *consonantal alphabet* is a limited set of phonograms from which words can be formed without vowels (1700BC)
- The modern alphabet dates to the Greek who added vowels to the alphabet (1000BC)

In comparison, formal languages and programming languages have evolved through time as well, however, much quicker!
• The Babylonians (3000–1500BC) introduced the base 60 notation for arithmetic and the earliest known algorithms.

• The Greek introduced proof. For example, Pythagoras (500BC) proved $c^2 = a^2 + b^2$ for the sides of a right triangle.

• The modern 10-digit numerical system dates back to Arabic scientists (780–300BC) and the number zero was introduced.

• Many contributed to the current mathematics: Gottlob Frege (1848–1925) (logic), Bertrand Russell (1872–1970) (formalization), David Hilbert (1861–1943) (constructive mathematics), and Alfred Tarski (1902–1983) (formal semantics).

Mathematics and logic are the tools of preference to study formal languages such as programming languages.
Alan M. Turing (1912–1954) introduced Turing machines, which are abstract machines that are potentially very powerful while having only a very limited number of operations. A Turing machine is a septuple \( \langle Q, T, I, \delta, b, q_0, q_f \rangle \), where

- \( Q \) is the set of states
- \( T \) is the set of tape symbols
- \( I \) is the set of input symbols, \( I \subseteq T \)
- \( \delta \) is the transition function
- \( b \in T \setminus I \) is the designated symbol for a blank
- \( q_0 \) is the initial state
- \( q_f \) is the final state

- **Lambda calculus** by Alonzo Church (Ch. 7)
- Konrad Zuse’s Plankalkül (1936)

\[
\begin{align*}
| & A + 1 \Rightarrow A \\
V & | 4 \quad 5 \\
S & | 1.n \quad 1.n
\end{align*}
\]

- **Flow diagrams** by Herman H. Goldstine and John von Neumann (1946)
• FORTRAN (FORmula TRANslating) (1957) is still used today and newer versions include parallelism (HPF)

• ALGOL (ALGOrichmic Language) (1960) has had a profound effect on programming language design

• LISP (LISt Processing) (late 50’s) primarily computes with lists of symbols rather than numeric data only

• SIMULA (early 60’s) was built on ALGOL as a system description language for discrete event network (simulation)

• COBOL (COmmon Business Oriented Language) (1960) influenced file system organization and files as collection of records

• APL (A Programming Language) (1966) has a very concise and compact notation for matrix/vector operations

• SNOBOL (“snowball”) (mid 60’s) primarily designed to process string data
In this course we will mainly focus on the principles of procedural and functional languages.
Object-Oriented (OO) languages are data-oriented, in contrast to procedure-oriented. Important features are:

- **object instances** and **classes**. A class encapsulates the type of data of an object with a set of **methods** that operate on an object instance

- **inheritance**: classes can inherit **data members** and **methods** from one other class or several classes (**multiple inheritance**)

Some example languages are:

- Smalltalk (1970) was heavily influenced by ideas introduced in SIMULA. In SIMULA, structures have a more independent existence and can operate on itself by a predetermined set of operations. SIMULA has a (limited) implementation of **abstract data types**, called **classes**. Smalltalk is the extreme of OOP. Even programming constructs are considered objects

- C++ (1982) is a popular OO language

- Java is like a stripped down version of C++. C++ is an inherently unsafe language, while Java is safe

Many language are not pure OO, but are a hybrid between imperative programming and OOP.
Declarative programming is an entirely different programming paradigm compared to imperative or procedural programming. Declarative programming languages are:

- Logic programming languages (Chapter 6: PROLOG)
- Functional programming languages (Chapter 7: lambda calculus)

PROLOG (PROgrammation en LOGique) is based on Robinson’s resolution technique for inference to deduce true formulas of predicate logic.

- PROLOG is nonprocedural: instead of specifying how a result should be computed, the properties of the result are specified
- PROLOG involves:
  - objects, consisting of terms (symbolic expressions)
  - facts about objects & relationships between objects
- PROLOG Example:

```prolog
man(socrates). % fact: object ‘socrates’ is a man
mortal(X) :- man(X). % rule: if X is a man, X is mortal
?- mortal(socrates). % query
yes % answer
?- mortal(zeus). % query
no % answer
?- mortal(X). % query
yes
  X = socrates % solution
; % more solutions?
no
```
Chapter 2 Syntax and Grammars

Chapter 2 topics:

- Programming Language Design
- Formal Languages
- Regular Expressions
- BNF
- Grammars and Chomsky Hierarchy
- Parsing
- Attribute Grammars and Static Semantics
- Post Systems
- Abstract Syntax
The criteria a well-designed programming language meets:

- **Abstraction**
  Factoring out recurring patterns. Examples are procedural abstraction and data abstraction.

- **Orthogonality**
  The basic features should be independent and separately understandable, and can be combined to build the control and data structures without exceptions. Examples are the construction of data types from primitive types. An example of a violation are functions in C that cannot return arrays.

- **Simplicity**
  The fewer concepts to understand a language the better.

- **Regularity**
  The only reasonable numbers are zero, one, and infinity. An example violation is that FORTRAN limits array dimensions to three.

- **Consistency**
  Similar constructs should look similar. An example violation is the notation `A(I)` for arrays which confuses them with function calls.

- **Translation**
  Can a compiler (quickly) generate efficient code for the language?
To study a programming language we have to study its

- **Syntax**
  The form.

- **Semantics**
  The meaning and interpretation. Often closely intertwined with syntax.

- **Pragmatics**
  Uses and effects of the language. For example, what type of hardware is required?

The syntax of a language has a profound effect on the ease of use of a language. Examples:

- The confusion between `=` and `==` in C.

- Block structure

  ```c
  if (i == 0) {
      printf("Stop\n");
      finish = 1;
  } else
      printf("i=%d\n", i);
  ``

- The possible omission of spaces in FORTRAN.

- The reserved keywords. E.g. in PL/I:

  ```c
  IF IF=THEN THEN THEN=ELSE; ELSE ELSE=IF
  IF IF THEN THEN=ELSE; ELSE ELSE=0
  ```

- One line, multi line, and nested comments?
Definitions:

- **Alphabet**
  Finite set of symbols. Examples: $\Sigma = \{0, 1\}$ and $\Sigma = \{\text{begin, end, :=, } x, 0\}$.

- **String (or sentence, word)**
  Finite sequence of symbols from the alphabet $\Sigma$. Examples: 1011 and $\text{begin } x := 0 \text{ end}$.

- **String length**
  Denoted $|s|$ for any string $s$.

- **Empty string**
  The special symbol $\epsilon$ (or ””). Note: $|\epsilon| = 0$.

- **Formal language**
  Set of strings over some fixed alphabet $\Sigma$. Examples:

  $$L_1 = \{00, 01, 10, 11\}$$

  $$L_2 = \{\epsilon, a, b, aa, ab, ba, bb,\text{ }
  \begin{array}{c}
  aaa, aab, aba, abb, baa, bab, bba, bbb
  \end{array}\}$$

  $$L_3 = \{x := 0, x := x, \text{begin } x := 0 \text{ end, } \ldots\}$$
We can combine languages in various ways to create new languages:

- **Union**
  \[ L \cup M = \{ s \mid s \in L \lor s \in M \} \]

- **Concatenation**
  \[ L \cdot M = \{ s \cdot t \mid s \in L \land t \in M \} \]

- **Exponentiation**
  \[ L^0 = \{ \epsilon \} \]
  \[ L^i = L \cdot L^{i-1} \]

- **Kleene closure**
  \[ L^* = \bigcup_{i=0}^{\infty} L^i \]

- **Positive closure**
  \[ L^+ = \bigcup_{i=1}^{\infty} L^i \]

For example, let \( \Sigma = \{0, 1, a, b\} \), \( L = \{a, b\} \), and \( M = \{0, 1\} \). Then
\[ L \cup M = \{0, 1, a, b\} \]
\[ L \cdot M = \{a0, a1, b0, b1\} \]
\[ L^2 = \{aa, ab, bb, ba\} \]
\[ L^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\} \]
Regular Expressions (REs) are a kind of shorthand for composing languages over the same alphabet. A regular expression is recursively defined as:

- **∅ (empty)**
  Denotes the language consisting of no strings, i.e. \( L = \emptyset \).

- **\( \epsilon \) (empty string)**
  Denotes the language consisting of the empty string, i.e. \( L = \{ \epsilon \} \).

- **a (atom)**
  Any symbol \( a \in \Sigma \) is a RE denoting \( L = \{ a \} \).

- **\( r_1 | r_2 \) (alternation)**
  If \( r_1 \) and \( r_2 \) are REs, \( r_1 | r_2 \) denotes the language that has all strings of the language denoted by \( r_1 \) and all strings of the language denoted by \( r_2 \).

- **\( r_1 r_2 \) (concatenation)**
  If \( r_1 \) and \( r_2 \) are REs, \( r_1 r_2 \) denotes the language that has all strings formed by concatenating a string of the language denoted by \( r_1 \) and a string of the language denoted by \( r_2 \).

- **\( r^* \) (closure)**
  If \( r \) is a RE, then \( r^* \) is a RE denoting the language formed by concatenating zero or more strings in the language denoted by \( r \).
We can make the meaning of regular expressions more formal by making the function more explicit. We call function $D$ “denotes” and it maps REs to languages:

\[
D[\emptyset] = \emptyset \\
D[\epsilon] = \{\epsilon\} \\
D[a] = \{a\} \quad \forall a \in \Sigma \\
D[r_1 \mid r_2] = D[r_1] \cup D[r_2] \\
D[r_1 \cdot r_2] = D[r_1] \cdot D[r_2] \\
D[r^*] = D[r]^*
\]

The mapping function above precisely defines the meaning of REs in terms of languages. E.g. let $\Sigma = \{a, b\}$. Then

\[
D[(a \mid b) (a \mid b)^*] = (\{a\} \cup \{b\}) \cdot (\{a\} \cup \{b\})^* \\
= \{a, b\} \cdot \bigcup_{i=0}^{\infty} \{a, b\}^i \\
= \{a, b, aa, ab, ba, bb, aaa, aab, \ldots\}
\]

We can easily see that $D[a \mid b] = \{a, b\}$ and $D[b \mid a] = \{a, b\}$, so we can prove that $(r_1 \mid r_2) = (r_2 \mid r_1)$ using the mapping $D$ and set theory. Other important identities are:

\[
\begin{align*}
(r_1 \mid r_2 \mid r_3) &= r_1 \cdot r_2 \mid r_1 \cdot r_3 \\
(r_1 \mid r_2) \cdot r_3 &= r_1 \cdot r_3 \cdot r_2 \cdot r_3 \\
\epsilon \cdot r &= r \\
\epsilon^* &= \epsilon \\
r \cdot \epsilon &= r \\
\emptyset^* &= \epsilon \\
\emptyset \cdot r &= \emptyset \\
(r \mid \epsilon)^* &= r^* \\
r \cdot \emptyset &= \emptyset \\
r^{**} &= r^*
\end{align*}
\]
Tools are available (e.g. Lex and Flex) that take a description of a set of regular expressions and generate recognizers for the REs. For designing a compiler for a programming language, this is particularly useful because the basic lexicographic entities of the language can be described using REs. For example, in Lex we can specify

```
  digit       [0-9]
  letter      [a-zA-Z]
  id          {letter}{letter}|{digit})*
  int         {digit}{digit}*
  [ \t\n]      { /* skip white space */ }
  "/\*"\(\n|.)\*/"   { /* skip comment */ }
  "/\//\."\n      { /* skip comment */ }
  ">="       { return LE; }
  [;,,=<>+-/*%(){}]   { return yytext[0]; }
  {id}        { return install_id(); }
  {int}       { return install_int(); }
```

This specification recognizes identifiers (id), integers (int), and operators (e.g. + and <=).

Basically, the RE recognizer is a deterministic finite automaton (DFA) with transitions on input characters and a final state that is reached after accepting a specific pattern. Also non-deterministic finite automata (NFA) can be constructed that are much smaller in size, but less efficient in recognizing patterns.
Backus-Naur Form (BNF) is a common notation to describe the syntax of a programming language.

In terms of BNF, a program is composed of a sequence of tokens. Tokens are the basic lexicographic entities of the language such as identifiers, numbers, keywords, punctuation, etc.

Tokens are the members of the alphabet of a programming language. Thus, a program is a string (sequence of tokens) of the programming language.

In BNF, the syntactic structure of the programming language is described by syntactic categories. For example, nested matching begin/end pairs can be described by

\[
\langle \text{nested} \rangle ::= \epsilon | \text{begin} \langle \text{nested} \rangle \text{ end}
\]

where

- “::=” means “is defined to be”.
- “⟨⟩” delimits syntactic categories.
- “|” means “or”.

Syntax of regular expressions over \( \Sigma = \{a, b\} \) in BNF:

\[
\langle RE \rangle ::= \emptyset | \epsilon | a | b \\
| ( \langle RE \rangle \ | \ | \langle RE \rangle ) \\
| ( \langle RE \rangle \langle RE \rangle ) \\
| \langle RE \rangle *
\]
Extended Backus-Naur Form (EBNF) is more expressive than BNF, because it includes repetition constructs.

- “::=” means “is defined to be”.
- “⟨ ⟩” delimits syntactic categories.
- “|” means “or”.
- “[ ]” means “optional”.
- “{ }” means “zero or more times”.
- “{ }+” means “one or more times”.

As an example, a Pascal compound statement in EBNF:

\[
\begin{align*}
\langle \text{compound} \rangle & ::= \text{begin} \langle \text{stmt} \rangle \{ ; \langle \text{stmt} \rangle \} \text{ end} \\
\langle \text{stmt} \rangle & ::= \text{if} \ (\langle \text{expr} \rangle) \ \text{then} \ \langle \text{stmt} \rangle \ [\text{else} \ \langle \text{stmt} \rangle]
\end{align*}
\]

BNF and EBNF are a special form of context-free grammars.
A grammar $G$ is a 4-tuple $\langle T, N, P, S \rangle$, where

- $T$ is the set of terminal symbols (or tokens).
- $N$ is the set of nonterminal symbols, $T \cap N = \emptyset$.
- $S$ is a start symbol, $S \in N$.
- $P$ is a finite set of pairs of the form $\alpha \rightarrow \beta$ the productions of the grammar, $\alpha \in (N \cup T)^* \cdot N \cdot (N \cup T)^*$ and $\beta \in (N \cup T)^*$.

Let $G$ be a grammar, then

- We say that $\gamma \alpha \delta \Rightarrow \gamma \beta \delta$ is a one-step derivation whenever $\alpha \rightarrow \beta$ is a production.
- We use
  \[ \alpha \Rightarrow^* \beta \]
  to denote the reflexive and transitive closure of the $\Rightarrow$ relation.
- A string $\omega$ of terminals is in the language defined by the grammar $G$ if
  \[ S \Rightarrow^* \omega \]
  In this case the string $\omega$ is called a sentence of $G$.
- We say that $\alpha$ is a sentential form of grammar $G$ if
  \[ S \Rightarrow^* \alpha \]
  where $\alpha$ may contain nonterminals.
The *Chomsky Hierarchy* relates the power of different types of grammars and language representations.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Machine(s)</th>
<th>Grammar constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unrestricted</td>
<td>Turing machines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post systems</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>context-sensitive</td>
<td>linear-bounded automata</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>context-free</td>
<td>pushdown automata, BNF</td>
<td>$\alpha \in N$</td>
</tr>
<tr>
<td>3</td>
<td>regular</td>
<td>finite automata, regular expressions</td>
<td>$A \rightarrow \omega B$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$A \rightarrow \omega$</td>
</tr>
</tbody>
</table>
Ch. 2  Grammars: an Example

- $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +\}$
- $N = \{\text{digit}, \text{digits}, \text{term}\}$
- $S = \text{term}$
- $P$ is the set of productions:
  
  \[
  \begin{align*}
  \text{term} & \rightarrow \text{term} + \text{digits} \\
  \text{term} & \rightarrow \text{digits} \\
  \text{digits} & \rightarrow \text{digit} \text{ digits} \\
  \text{digits} & \rightarrow \text{digit} \\
  \text{digit} & \rightarrow 0 \\
  \text{digit} & \rightarrow 1 \\
  \text{digit} & \rightarrow \ldots \\
  \text{digit} & \rightarrow 9
  \end{align*}
  \]

In the following derivation, each leftmost nonterminal is replaced by a right-hand side of a production of the nonterminal:

\[
\begin{align*}
S & \Rightarrow \text{term} + \text{digits} \\
& \Rightarrow \text{digits} + \text{digits} \\
& \Rightarrow \text{digit} \text{ digits} + \text{digits} \\
& \Rightarrow 1 \text{ digits} + \text{digits} \\
& \Rightarrow 1 \text{ digit} + \text{digits} \\
& \Rightarrow 19 + \text{digits} \\
& \Rightarrow 19 + \text{digit} \\
& \Rightarrow 19 + 0
\end{align*}
\]
Sentence $\omega$ is in the language defined by a grammar if

$$S \Rightarrow^* \omega$$

In the example, we saw that

$$S \Rightarrow^* 19 + 0$$

The string $19 + +0$ cannot be derived from the grammar and is therefore not in the language defined by the grammar: $19 + +0$ is syntactically incorrect with respect to the grammar. The problem is, how can we efficiently recognize when a string is a sentence of a grammar? This is called the parsing problem. A parser attempts to recognize a string as a sentence of the grammar by constructing the reverse derivation:

$$1 9 + 0$$

$$digit 9 + 0$$

$$digit digit + 0$$

$$digit digits + 0$$

$$digits + 0$$

$$term + 0$$

$$term + digit$$

$$term + digits$$

$$term$$

$$S$$

Note that this is not an easy task, because we have to make sure that we select the “right” nonterminal to replace a sequence of grammar symbols in each of the (intermediate) sentential forms.
A parse tree of a sentence depicts its grammatical structure.

- The root of the tree consists of the start symbol.
- Each node in the parse tree consists of a nonterminal.
- Each leaf in the parse tree consists of a terminal.
- The edges correspond to the right-hand sides of productions.

For example, the parse tree of \(19 + 0\) is:
Ch. 2  *Ambiguous Grammars*

A grammar is *ambiguous* if a sentence has more than one derivation and therefore has more than one interpretation leading to confusion.

Consider for example the grammar

\[
S \rightarrow \text{if } C \text{ then } S \\
\text{if } C\text{ then } S \text{ else } S \\
S'
\]

Then the string \texttt{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2 \text{ has two reasonable interpretations indicated by the parse trees}

![Parse tree 1](image1)

![Parse tree 2](image2)
We distinguish two types of semantics of a programming language:

- **Static semantics** describe specific constraints of a programming language that can be checked at compile-time. Examples are type checking, variable declaration and variable use checking, uniqueness of function declarations, etc.

- **Dynamic semantics** describe the dynamic “behavior” of a program at run-time (main topic of Chapters 8 and 9).

As an example of static semantics, consider the following (simplified) EBNF of the ADA block statement syntax:

\[
\langle \text{block} \rangle ::= \langle \text{id} \rangle : \text{begin} \langle \text{stmt} \rangle \{\langle \text{stmt} \rangle \} \text{end} \langle \text{id} \rangle ;
\]

ADA restricts that the first and second block identifiers must be identical. This could be expressed in a context-sensitive grammar. As a result, however, an efficient parser for the language cannot be constructed.

Instead, the constraint might be expressed using values associated with grammar symbols (terminals and nonterminals). For example, the constraint can be checked by asserting that

\[
\langle \text{id} \rangle_1.\text{value} = \langle \text{id} \rangle_2.\text{value}
\]

where the values hold the actual names of the first \(\langle \text{id} \rangle_1\) and second \(\langle \text{id} \rangle_2\) identifiers.
An attribute grammar is a context-free grammar augmented with attributes and semantic rules.

Attributes hold values of terminals and nonterminals and are generally written as $x.a$ where $x$ is a terminal or nonterminal and $a$ is an attribute name.

Semantic rules are associated with grammar productions and consist of assignments of values to attributes. For example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \rightarrow id_1: \text{begin } S \text{ end } id_2;$</td>
<td>$B.ok := id_1.val = id_2.val$</td>
</tr>
</tbody>
</table>

An annotated parse tree is a parse tree showing the values of attributes. For example

```
<table>
<thead>
<tr>
<th>B.ok=true</th>
</tr>
</thead>
<tbody>
<tr>
<td>id.val=&quot;go&quot;</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>id.val=&quot;go&quot;</td>
</tr>
<tr>
<td>go : begin</td>
</tr>
</tbody>
</table>
```
• **Synthesized attributes** of a nonterminal are set in the semantic rule of a production of the nonterminal

\[ A \rightarrow X_1 \ldots X_n \quad A.s := f(A.a, X_1.a, \ldots, X_n.a) \]

where \( f \) is a function and \( X_k.a \, (k = 1, \ldots, n) \) are attributes of grammar symbols \( X \). The values of synthesized attributes are propagated upwards in a parse tree:

![Parse Tree Diagram]

• **Inherited attributes** of a nonterminal are set in the semantic rule of a production of a parent nonterminal

\[ A \rightarrow X_1 \ldots X_n \quad X_j.i := f(A.a, X_1.a, \ldots, X_n.a) \]

for some \( j, \, (1 \leq j \leq k) \) where \( f \) is a function and \( X_k.a \, (k = 1, \ldots, n) \) are attributes of grammar symbols \( X \). The values of inherited attributes are propagated downwards in a parse tree:

![Parse Tree Diagram]
Here is the “terms and digits” grammar again in a more compact notation, now as an attribute grammar for calculating sums of integers:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1 \rightarrow T_2 + N$</td>
<td>$T_1.val := T_2.val + N.val$</td>
</tr>
<tr>
<td>$T \rightarrow N$</td>
<td>$T.val := N.val$</td>
</tr>
<tr>
<td>$N_1 \rightarrow D N_2$</td>
<td>$N_1.pos := N_2.pos + 1$</td>
</tr>
<tr>
<td></td>
<td>$N_1.val := D.val + N_2.val$</td>
</tr>
<tr>
<td></td>
<td>$D.pow := N_2.pos$</td>
</tr>
<tr>
<td>$N \rightarrow D$</td>
<td>$N.pos := 1$</td>
</tr>
<tr>
<td></td>
<td>$N.val := D.val$</td>
</tr>
<tr>
<td></td>
<td>$D.pow := 0$</td>
</tr>
<tr>
<td>$D \rightarrow 0$</td>
<td>$D.val := 0$</td>
</tr>
<tr>
<td>$D \rightarrow 1$</td>
<td>$D.val := 10^{D.pow}$</td>
</tr>
<tr>
<td>$D \rightarrow 9$</td>
<td>$D.val := 9 \times 10^{D.pow}$</td>
</tr>
</tbody>
</table>

We see that

<table>
<thead>
<tr>
<th>Synthesized</th>
<th>Inherited</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T.val$</td>
<td></td>
</tr>
<tr>
<td>$N.pos$</td>
<td></td>
</tr>
<tr>
<td>$N.val$</td>
<td></td>
</tr>
<tr>
<td>$D.val$</td>
<td>$D.pow$</td>
</tr>
</tbody>
</table>
The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

This graph determines the parse tree traversal to evaluate the attributes.

The graph may not be cyclic, otherwise we cannot find an evaluation order.
Production | Semantic Rule
---|---
\(S \rightarrow E\) | \(E.env := \text{NIL}\)
\(E \rightarrow 0\) | \(E.val := 0\)
\(E \rightarrow 1\) | \(E.val := 1\)
\(E \rightarrow \text{id}\) | \(E.val := \text{LookUp}(\text{id}.name, E.env)\)
\(E_1 \rightarrow \text{let id = } E_2 \text{ in } E_3 \text{ end}\) 
\(E_2.env := E_1.env\)
\(E_3.env := \text{Update}(E_1.env, \text{id}.name, E_2.val)\)
\(E_1.val := E_3.val\)
\(E_1 \rightarrow ( E_2 + E_3 )\)
\(E_1.val := E_2.val + E_3.val\)
\(E_2.env := E_1.env\)
\(E_3.env := E_1.env\)

Example annotated parse tree:
Ch. 2 Abstract Syntax

• A parse tree is sometimes called a concrete syntax tree

• An abstract syntax tree (AST) is like a parse tree, but the terminals and other “uninteresting” details have been removed. Instead, the nodes of the tree consist of constructors.

Consider for example the grammar defining the concrete syntax of a virtual language

\[ S \rightarrow \text{id} \:=\ E \]

<table>
<thead>
<tr>
<th>\hspace{1cm}</th>
<th>\hspace{1cm}</th>
<th>\hspace{1cm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{if } E \text{ then } S \text{ else } S</td>
<td>\text{while } E \text{ do } S</td>
<td>S ; S</td>
</tr>
</tbody>
</table>

In an abstract syntax, we are not interested in

• the form of the keywords and tokens, e.g. := may be something else

• whether semicolons are separators or terminators

• ambiguity of the grammar

We assume that all of these issues are resolved by a parser. The four constructs are represented by four constructors, respectively

\[
\begin{align*}
\text{assign}(id, E) \\
\text{cond}(E, S_1, S_2) \\
\text{loop}(E, S_1) \\
\text{block}(S_1, S_2)
\end{align*}
\]
The four constructors are used to compose abstract syntax trees. For example, consider the program \texttt{while }E \texttt{ do } S_1 ; S_2. Since the grammar is ambiguous, there are two parse trees for this program.

The ambiguous program can be rendered in two ways

representing the abstract syntax trees. Once an abstract syntax tree has been created, it can be easily manipulated, e.g. by a compiler.
Post systems are useful deductive formal systems for reasoning about programming languages.

A Post system consists of

- A list of signs that forms the alphabet of the system. A string of signs is called a word.
- A list of variables. A string of signs and variables is called a term.
- A finite set of productions of the form

\[
\frac{\text{premises}}{\text{conclusion}} \quad \frac{t_1 \ t_2 \ \cdots \ t_n}{t}
\]

where the \( t, t_1, \ldots, t_n \ (n \geq 0) \) are terms. A production without premises \((n = 0)\) is called an axiom.

An instance of a production is obtained by substituting words for all the variables, the same word being substituted for one and the same variable. For example

\[
\frac{ax \ bxy}{yax} \quad \text{subst. } x = c \text{ and } y = ab \quad \frac{ac \ bcab}{abac}
\]
The set of words that are provable from a Post system forms the language derived from the system.

The proof of a word is inductively defined as

- An instance of an axiom, e.g.

  \[
  \begin{array}{c}
  \hline
  a \\
  \hline
  \end{array}
  \]

  is a proof of the conclusion, e.g. word \( a \).

- If \( Pr_1, Pr_2, \ldots, Pr_n \) are proofs of the terms \( a_1, a_2, \ldots, a_n \), respectively, and

  \[
  \begin{array}{cccc}
  \hline
  a_1 & a_2 & \cdots & a_n \\
  \hline
  a \\
  \end{array}
  \]

  is an instance of a production, then

  \[
  \begin{array}{cccc}
  \hline
  Pr_1 & Pr_2 & \cdots & Pr_n \\
  \hline
  a \\
  \end{array}
  \]

  is a proof of \( a \).

Since all \( Pr_i \) are proofs of the form

\[
\begin{array}{c}
\hline
p_i \\
\hline
a_i \\
\end{array}
\]

the “combined” proof above is written as

\[
\begin{array}{cccc}
\hline
p_1 & p_2 & \cdots & p_n \\
\hline
a_1 & a_2 & \cdots & a_n \\
\hline
a \\
\end{array}
\]
Post Systems: Example 1

Tally notation writes a natural number \( n \geq 0 \) as \( N \) followed by a sequence of \( | \) of length \( n \).

Consider the Post system with signs \( \{N, |\} \), one variable \( x \), and two productions

\[
\begin{align*}
\overline{N} & \quad \frac{N \ x}{N \ x \ |} \\
\end{align*}
\]

- \( N \) is a word of the language derived by the Post system, since we have the instance of the axiom

\[
\overline{N}
\]

- \( N | \) is a provable word, because

\[
\begin{align*}
\overline{N} & \quad \frac{N}{N |}
\end{align*}
\]

is an instance, and we have the proof of \( N \). Combining the proofs yields

\[
\begin{align*}
\overline{N} & \quad \frac{N}{N |}
\end{align*}
\]

- By repeatedly combining proofs, we find that the Post system derives words of the form \( N || \ldots | \)
The following Post system derives addition equations of the form $x + y = z$. The Post system has signs $\{N,|,+,-\}$, variables $x$, $y$, and $z$, and the four productions

\[
\begin{align*}
N & \rightarrow Nx \\
N & \rightarrow Nx | \\
N y & \rightarrow N y + y = y \\
x + y & \rightarrow x | + y = z \\
\end{align*}
\]

Here is the proof of $|| + ||=||| (i.e. \ 2 + 2 = 4)$

\[
\begin{align*}
N & \rightarrow N | \\
N & \rightarrow N || \\
+ & \rightarrow ||=|| \\
| + & \rightarrow ||=||| \\
|| + & \rightarrow |||=||| \\
\end{align*}
\]

Productions in Post systems may have more than one premise. As an example, reconsider the previous example Post system, now with three productions

\[
\begin{align*}
N & \rightarrow Nx \\
N x & \rightarrow Nx | \\
N x & \rightarrow N x N y \\
x + y & \rightarrow x y \\
\end{align*}
\]

The formula $x + y = xy$ makes use of the fact that the $|$’s can be concatenated to form the sum. An example proof is

\[
\begin{align*}
N & \rightarrow N| \\
N & \rightarrow N| N \\
| + & \rightarrow =| \\
\end{align*}
\]
Post systems and grammars are formal systems. In fact, Post systems are formalisms for unrestricted grammars that describe type 0 languages.

A comparison of common terminology for formal systems:

<table>
<thead>
<tr>
<th>Formal system</th>
<th>Grammar</th>
<th>Post System</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>alphabet</td>
<td>signs</td>
</tr>
<tr>
<td>well-formed formula</td>
<td>string</td>
<td>word</td>
</tr>
<tr>
<td>axiom</td>
<td>start symbol</td>
<td>axiom</td>
</tr>
<tr>
<td>rule of inference</td>
<td>production</td>
<td>production</td>
</tr>
<tr>
<td>theorem</td>
<td>sentential form</td>
<td>term in proof</td>
</tr>
</tbody>
</table>
As an application of Post systems, we will consider type inference in languages with *parametric polymorphism* (see Section 4.5.3).

- *Parametric* means that types can be parameterized.
- *Polymorphic* means that expressions can have more than one type.

We consider the mini-ML functional programming language as an example. Its syntax is

\[
\langle expr \rangle ::= \langle id \rangle \\
| ( \text{fn} \langle id \rangle \Rightarrow \langle expr \rangle ) \\
| \langle expr \rangle \langle expr \rangle \\
| ( \text{if} \langle expr \rangle \text{then} \langle expr \rangle \text{else} \langle expr \rangle )
\]

For example, \((\text{fn } x \Rightarrow \text{sin } x)\) 1 evaluates to \(\text{sin}(1) = 0.841470984807897\).

A *type* is an expression defined by

\[
\langle type \rangle ::= \langle type\_variable \rangle \\
| \text{int} \\
| \text{real} \\
| \text{bool} \\
| \langle type \rangle \Rightarrow \langle type \rangle
\]

\[
\langle type\_variable \rangle ::= \' \langle id \rangle
\]

For example, the type of \((\text{fn } x \Rightarrow \text{sin } x)\) is \(\text{real} \Rightarrow \text{real}\) and the type of \((\text{fn } x \Rightarrow x)\) is \(\text{'a} \Rightarrow \text{'a}\) meaning that it is a mapping from e.g. \text{int} \Rightarrow \text{int}, \text{real} \Rightarrow \text{real}, etc.
Typing rules for mini-ML are given by a Post system that will be used to separate “good” strings (correctly typed expressions) from “bad” strings (incorrectly typed expressions).

We will call strings derived from the system judgements. A typing judgement has the form

$$ A \vdash e : \tau $$

where $A$ is a type assignment consisting of a set of pairs binding identifiers to types, $e$ is an expression, $\tau$ is a type. We say that “$e$ has type $\tau$ under assignment $A$”.

The following four productions give typing rules for the four programming constructs:

1. If $x \in \text{Dom}(A)$
   $$ A \vdash x : A(x) $$

2. $$ A \vdash \text{fn } x \rightarrow e_1 : \tau_1 \rightarrow \tau_2 $$
   $$ A \vdash e_1 : \tau_1 \rightarrow \tau_2 $$

3. $$ A \vdash e_1 : \tau_1 \rightarrow \tau_2 $$
   $$ A \vdash e_2 : \tau_1 $$
   $$ A \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau $$

where $x \in \text{Dom}(A)$ is true when identifier $x$ is paired with a type in $A$ in which case $A(x)$ returns $x$’s type. $A[x \mapsto \tau]$ denotes $A$ updated with the pair $\langle x, \tau \rangle$. 
Type checking can be viewed as proving theorems in the Post system defined by the four rules.

Consider for example the expression

\[(\text{fn } x \Rightarrow (\text{if } \text{true} \text{ then } 0 \text{ else } 1))\]

Let \(A_0 = \{\langle \text{true}, \text{bool} \rangle, \langle 0, \text{int} \rangle, \langle 1, \text{int} \rangle\}\) be the initial set of type assignments.

Then, the following proof confirms that the type of the example expression is \(\texttt{'}a \rightarrow \text{int}'\):

\[
\begin{align*}
A_1 \vdash \text{true} : \text{bool} & \quad A_1 \vdash 0 : \text{int} & \quad A_1 \vdash 1 : \text{int} \\
A_1 \vdash (\text{if } \text{true} \text{ then } 0 \text{ else } 1) : \text{int} & \\
A_0 \vdash (\text{fn } x \Rightarrow (\text{if } \text{true} \text{ then } 0 \text{ else } 1)) : \texttt{'}a \rightarrow \text{int}
\end{align*}
\]

where

\[
A_1 = A_0[x \mapsto \texttt{'}a] \\
= \{\langle x, \texttt{'}a \rangle, \langle \text{true}, \text{bool} \rangle, \langle 0, \text{int} \rangle, \langle 1, \text{int} \rangle\}
\]

Note that type variables (e.g. \(\texttt{'}a\)) can take an infinite number of types.

If we cannot find a proof, the expression is incorrectly typed. For example \((\text{if } \text{true} \text{ then } \text{true} \text{ else } 0)\). Here, \text{true} and 0 have different types.
More practically speaking, when type inference takes place the choices of an identifier’s type are determined from its use in an expression.

\[
(f : x = \rightarrow (\text{if } \text{true} \text{ then } x \text{ else } 1))
\]

\[
\frac{A_1 \vdash \text{true} : \text{bool} \hspace{1cm} A_1 \vdash x : 'a \hspace{1cm} A_1 \vdash 1 : \text{int}}{A_1 \vdash (\text{if } \text{true} \text{ then } x \text{ else } 1) : \text{int}}
\]

\[
A_0 \vdash (f : x = \rightarrow (\text{if } \text{true} \text{ then } x \text{ else } 1)) : \text{int} \rightarrow \text{int}
\]

with 'a = int being determined when the rule for the conditional expression is applied.

An algorithm like this “instantiation scheme for type variables” is implemented as TypeOf(A, e). Milner (1978) has proven

- **Correctness:** If TypeOf(A, e) succeeds with τ, then A ⊢ e : τ is derivable from the typing rules.

- **Completeness:** If A ⊢ e : σ is derivable from the typing rules, then TypeOf(A, e) succeeds with τ and σ is an instance of τ.
Read Chapter 2 and Section 4.5 (especially 4.5.3). You may skip Section 2.4.4.

The rules to change EBNF into BNF without “or” are

- productions of the form $\langle X \rangle ::= \alpha \mid \beta$ are changed into $\langle X \rangle ::= \alpha$ $\langle X \rangle ::= \beta$

- productions of the form $\langle X \rangle ::= \alpha [\beta] \gamma$ are changed into
  $\langle X \rangle ::= \alpha \gamma$
  $\langle X \rangle ::= \alpha \beta \gamma$

- productions of the form $\langle X \rangle ::= \alpha \{\beta\} \gamma$ are changed into
  $\langle X \rangle ::= \alpha \langle Y \rangle \gamma$
  $\langle Y \rangle ::= \beta \langle Y \rangle$ where $Y$ is a new nonterminal.
  $\langle Y \rangle ::= \epsilon$
Why is lambda calculus important to computer science?

- It is a theoretical model of computation. Equally powerful as Turing machines, but easier to program.
- It captures the functional programming paradigm of all functional languages.
- Functions are first-class objects in functional languages: functions can be given as arguments to functions and returned by functions. Only a few procedural languages treat functions first class.
- Its notation distinguishes between a function and the value of a function, which are often confused: $f(x)$ is not a function, but $f$ is.
- Working with lambda calculus is like playing a game. The syntactic notation is very simple, but it has a powerful meaning that may be puzzling at first.
A function is a relation that associates each element of a given type $A$ a unique member of a second type $B$, which can be expressed as

$$f :: A \rightarrow B$$

$A$ is called the source type and $B$ is called the target type.

If $x$ denotes an element of $A$, we write $f(x)$ or just $f x$ to denote the result of applying function $f$ to $x$.

Now, consider the following ANSI C fragment

```c
int add(int y, int z) { return y+z; }
int inc(int x) { return add(1, x); }
```

Functions $\text{add}$ and $\text{inc}$ can be expressed mathematically as

$$\text{add} :: (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$$
$$\text{add} :: (y, z) \mapsto y + z$$

$$\text{inc} :: \mathbb{Z} \rightarrow \mathbb{Z}$$
$$\text{inc} :: x \mapsto \text{add}(1, x)$$

Notice that the “function body” of $\text{inc}$ is $x \mapsto \text{add}(1, x)$, denoting the mapping of element $x \in \mathbb{Z}$ to $\text{add}(1, x) \in \mathbb{Z}$.

Lambda calculus functions are anonymous mappings. The $\text{inc}$ function is written in lambda calculus as

$$\lambda x. \text{add} \ 1 \ x$$

called an abstraction. Notice that we omit parenthesis and commas, as these are not essential for understanding the meaning of the function.
• In the abstraction

$$\lambda x. \text{add } 1 \ x$$

$x$ is a formal argument, just like the $x$ in

```c
int inc(int x) { return add(1, x); }
```

Renaming the formal argument $x$ in the `inc` function to $v$ does not change the meaning of the function. Likewise, the abstraction

$$\lambda v. \text{add } 1 \ v$$

has exactly the same meaning as the previous expression. This equivalence is called alpha equivalence, and alpha reduction is the renaming of formal arguments of abstractions.

• The application of an abstraction to an actual argument, say 6, is written as

$$(\lambda x. \text{add } 1 \ x) \ 6$$

But what does this expression mean? Basically, we apply the abstraction on 6 by substituting the actual argument 6 for the formal argument $x$. We obtain

$$\text{add } 1 \ 6$$

This is called beta reduction.
A higher-order function is a function that takes a function as an argument or returns a (new) function. Returning new functions is not possible in C/C++. Consider:

```c
int(int) add(int y) {
    int partial_add(int z)
    { return y+z; }
    return partial_add;
}
```

`add` is not a legal C function. It is intended to take a first argument and return another function that takes the second argument. Here is a hypothetical evaluation:

```c
add(1)(6)  
⇒ partial_add(6)  
⇒ 1 + 6  
⇒ 7
```

Here, formal argument `y` takes value `1` which is substituted in the function body of `partial_add` which is returned by `add`. Then, `partial_add` takes `6`, adds it to `1` and returns `7`. This hypothetical evaluation in C explains why we can write the add function in lambda calculus as

\[ \lambda y. \lambda z. y + z \]

Consider the application

\[ (\lambda y. \lambda z. y + z) \ 1 \ 6 \]

Assuming application is left associative, beta reduction yields

\[ (\lambda z. 1 + z) \ 6 \]

After another beta reduction we obtain `1 + 6`. 
Lambda expressions are generally written according to the following (ambiguous) grammar

\[
V \to x \mid y \mid z \mid u \mid v \mid \ldots \\
F \to a \mid b \mid c \mid f \mid g \mid \ldots \\
L \to V \mid F \mid \lambda V . L \mid LL \mid (L)
\]

where \(V\) are the variables, \(F\) are the function symbols, and \(L\) lambda expressions. Expressions of the form \(\lambda V . L\) are called abstractions and \(LL\) are called applications.

To disambiguate the grammar, we follow two rules:

1. Application has precedence over abstraction.
   For example, we write \((\lambda x . f x) y\) for \((\lambda x . (f x)) y\).

2. Application is left associative.
   For example, we write \(f a x\) for \((f a) x\).

Here are some examples of lambda expressions

\[
\begin{align*}
\lambda x . x \\
f x y \\
f (g x) \\
f x (g y) \\
f (\lambda x . g x) \\
(\lambda x . b) (f y) \\
(\lambda x . g z) (f y) \\
\lambda x . \lambda y . f x y
\end{align*}
\]

Exercise: insert the proper parenthesis according to the disambiguation rules.
Let $L$ be a lambda expression. The set of free variables of $L$ denoted $FV[L]$ is defined as

- $FV[V] = \{V\}$ for variable $V$.
- $FV[F] = \emptyset$ for function symbol $F$.
- $FV[L_1 \, L_2] = FV[L_1] \cup FV[L_2]$ for application $L_1 \, L_2$.

We say that lambda expression $L$ is closed if $FV[L] = \emptyset$.

A variable can appear both free and bound in an expression. For example, in

$$(\lambda x . x) \, x$$

the rightmost $x$ is free, while the variable $x$ in $\lambda x . x$ is bound.

Examples

- $f \, x \, y$ \quad $FV = \{x, y\}$
- $f \, (g \, x)$ \quad $FV = \{x\}$
- $f \, x \, (g \, y)$ \quad $FV = \{x, y\}$
- $f \, (\lambda x . g \, x)$ \quad $FV = \emptyset$
- $(\lambda x . b) \, (f \, y)$ \quad $FV = \{y\}$
- $(\lambda x . g \, z) \, (f \, y)$ \quad $FV = \{y, z\}$
- $\lambda x . \lambda y . f \, x \, y$ \quad $FV = \emptyset$
When we apply an abstraction to an argument (beta reduction), the formal argument (a variable) in the body of the abstraction is replaced by the actual argument (an expression).

For example:

\[(\lambda x. f a x) 3 \Rightarrow f a 3\]

The *clash of variables problem*: consider expression

\[(\lambda x. \lambda y. f x y) \ (g y)\]

where \(y\) is free. When we apply beta reduction and substitute \(g y\) for \(x\) in the abstraction we obtain

\[\lambda y. f \ (g y) \ y\]

Variable \(y\) of \(g y\) is accidentally bound in the abstraction!

Clearly, this is an invalid substitution. We can avoid this by renaming \(y\) by e.g. \(z\) in the abstraction (i.e. alpha reduction)

\[(\lambda x. \lambda z. f x z) \ (g y)\]

When we substitute \(g y\) for \(x\) we obtain

\[\lambda z. f \ (g y) \ z\]

in which the variable clash problem has been avoided.
Let \( t \) and \( e \) be lambda expressions. The function for substituting \( t \) for a variable \( x \) in \( e \) denoted \( e[x := t] \) is recursively defined as follows:

1. \( x[x := t] := t \).
2. \( v[x := t] := v \) for variable \( v \) and \( x \neq v \).
3. \( f[x := t] := f \) for function symbol \( f \).
4. \( (\lambda x . b)[x := t] := \lambda x . b \).
5. \( (\lambda v . b)[x := t] := \lambda v . b \) if \( x \neq v \) and \( x \not\in FV[b] \).
6. \( (\lambda v . b)[x := t] := \lambda v . b[x := t] \) if \( x \neq v \) and \( v \not\in FV[t] \).
7. \( (\lambda v . b)[x := t] := \lambda z . (b[v := z])[x := t] \) if \( x \neq v \) and \( v \in FV[t] \), where \( z \) is a new variable such that \( z \not\in FV[t] \) and \( z \not\in FV[b] \).
8. \( (\ell m)[x := t] := (\ell[x := t]) \ (m[x := t]) \).

Consider e.g. \( (\lambda x . \lambda y . f \ x \ y) \ (g \ y) \). The substitution

\[
(\lambda y . f \ x \ y)[x := g \ y]
\]

needs renaming (rule 7), because \( y \in FV[g \ y] = \{y\} \)

\[
(\lambda z . (f \ x \ y)[y := z])[x := g \ y]
\]

where \( z \) is a new variable. Then (rules 1, 2, 3, and 8)

\[
(\lambda z . f \ x \ z)[x := g \ y]
\]

Rule 6 gives \( \lambda z . (f \ x \ z)[x := g \ y] \) and finally \( \lambda z . f \ (g \ y) \ z \).
The substitution algorithm provides a mechanism for evaluating lambda expressions. That is, a meaning is assigned to a lambda expression of the form

\[(\lambda v . b) \ t\]

in that the substitution \(b[v := t]\) is applied resulting in the replacement of a lambda expression by another.

However, an order in evaluating lambda expressions has not been defined yet. Consider for example

\[(\lambda x . x^2) \ ((\lambda y . y + 1) \ 3)\]

We can either apply the leftmost abstraction, giving

\[((\lambda y . y + 1) \ 3)^2\]

and then

\[4^2 = 16\]

Or we can apply the rightmost abstraction, giving

\[(\lambda x . x^2) \ 4\]

and then

\[4^2 = 16\]

Both results are identical. As it turns out, the order of application is immaterial (except for the possibility of non-terminating sequences). To prove this in the general case, we have to resort to methods from term rewriting systems (TRS).
The alpha and beta reductions in lambda calculus are applications of rewrite rules on lambda expressions. A relation $>_{\text{rel}}$ is called a contraction if it relates two lambda expressions $L$ and $M$

$$L > M$$

For example, we can define an example contraction

$$f > \lambda x . x$$

which relates redex $f$ with contractum $\lambda x . x$.

The alpha, beta, and eta contractions are defined as

- **\( \alpha \) contraction:**

  $$\lambda V . L >_{\alpha} \lambda W . L[V := W]$$ for all lambda expressions $L$ and variables $V$ and $W$ such that $W \notin FV[L]$.

- **\( \beta \) contraction:**

  $$(\lambda V . L) M >_{\beta} L[V := M]$$ for all lambda expressions $L$ and $M$ and variables $V$.

- **\( \eta \) contraction:**

  $$\lambda V . LV >_{\eta} L$$ for all lambda expressions $L$ and variables $V$ such that $V \notin FV[L]$. 
The one-step reduction relation $\rightarrow$ on lambda expressions is defined as

$$L \rightarrow M \quad \text{if} \quad L > M$$

(5)

$$\lambda V . L \rightarrow \lambda V . M \quad \text{if} \quad L \rightarrow M \quad \text{for all variables} \ V$$

(6)

$$L N \rightarrow M N \quad \text{if} \quad L \rightarrow M$$

(7)

$$N L \rightarrow N M \quad \text{if} \quad L \rightarrow M$$

(8)

For lambda expressions $L$, $N$, and $M$, and variables $V$.

For example

$$(\lambda v . f \ v) \ x \rightarrow_\beta f \ x$$

(one-step beta reduction using rule 1)

$$(\lambda x . f \ x) \ x \rightarrow_\alpha (\lambda v . f \ v) \ x$$

(one-step alpha reduction using rule 3 and 1)

$$(\lambda x . (\lambda y . y \ x) \ f) \ x \rightarrow_\beta (\lambda x . f \ x) \ x$$

(one-step beta reduction using rules 3, 2 and 1)

The reflexive, transitive closure of the one-step relation $\rightarrow$ is the reduces relation:

$$\Rightarrow$$

defined as

$$L \Rightarrow L$$

(9)

$$L \Rightarrow M \quad \text{if} \quad L \rightarrow M$$

(10)

$$L \Rightarrow N \quad \text{if} \quad L \Rightarrow M \ \text{and} \ M \Rightarrow N$$

(11)

For lambda expressions $L$, $N$, and $M$. 
Ch. 7  Alpha Equivalence

The reflexive, symmetric, and transitive closure of a contraction, denoted $\equiv$, is defined as

\begin{align*}
L \equiv L & \quad (12) \\
L \equiv M & \text{ if } L \to M \quad (13) \\
L \equiv M & \text{ if } M \equiv L \quad (14) \\
L \equiv N & \text{ if } L \equiv M \text{ and } M \equiv N \quad (15)
\end{align*}

For lambda expressions $L$, $M$, and $N$.

**Theorem**
The relations

$$\star \to_\alpha$$

and

$$\equiv_\alpha$$

are identical.

**Proof**
The alpha-contraction relation is symmetric:

$$\lambda V . L \ >_\alpha \lambda W . L[V := W]$$

$$\quad >_\alpha \lambda V . (L[V := W])[W := V]$$

$$\quad = \lambda V . L$$

Therefore, the relations are symmetric.  

The equivalence relation $\equiv_\alpha$ forms equivalence classes of lambda expressions. Two lambda expressions $L$ and $M$ are *alpha equivalent* (also called *alphabetically equivalent*) if $L \equiv_\alpha M$. Basically, $L$ and $M$ are identical up to the choice of names for the bound variables.
Beta reduction is the process of repeatedly replacing beta redexes. The selection and replacement of a beta redex continues until no beta redex remains in the expression.

A lambda expression is in \textit{beta normal form} if it has no beta redexes.

Examples:

\begin{itemize}
  \item $f$ is in normal form
  \item $\lambda x . x$ is in normal form
  \item $(\lambda x . x) \ y$ is \textit{not} in normal form
  \item $\lambda x . \lambda y . y$ is in normal form
  \item $\lambda x . (\lambda y . y) \ 1$ is \textit{not} in normal form
\end{itemize}

This immediately suggests a naive evaluation scheme:

\begin{verbatim}
while there are more redexes do
  reduce one of the redexes
od
\end{verbatim}

{ expression is now in normal form }

Hence, a beta normal form of a lambda expression represents the “end result of a computation”.

But is the choice of redex important? Does this evaluation scheme always terminate? Is the result always the same?
Consider the following two distinct beta reductions of the same lambda expression

\[(\lambda x. \lambda y. y) \ ((\lambda z. z) \ (\lambda z. z))\]

\[\rightarrow_{\beta} \lambda y. y\]

and

\[(\lambda x. \lambda y. y) \ ((\lambda z. z) \ (\lambda z. z))\]

\[\rightarrow_{\beta} (\lambda x. \lambda y. y) \ ((\lambda z. z) \ (\lambda z. z))\]

\[\rightarrow_{\beta} (\lambda x. \lambda y. y) \ ((\lambda z. z) \ (\lambda z. z))\]

\[\rightarrow_{\beta} \vdots\]

Here, the redex on the right \((\lambda z. z) \ (\lambda z. z)\) is reduced first, leading to an infinite sequence of reductions.

This example shows that the selection of a redex to reduce is important.
We distinguish the following redexes in an expression:

- The *leftmost* redex is the redex whose $\lambda$ is textually to the left of all other redexes within the expression.

- The *rightmost* redex is the redex whose $\lambda$ is textually to the right of all other redexes within the expression.

- The *outermost* redex is a redex which is not contained within any other redex.

- The *innermost* redex is a redex which contains no other redex.

There are two important reduction orders:

- **Applicative-order reduction** (AOR)
  AOR reduces the *leftmost innermost* redex first.
  c.f. call-by-value argument passing.

- **Normal-order reduction** (NOR)
  NOR reduces the *leftmost outermost* redex first.
  c.f. call-by-name argument passing.

In the example

$$\lambda x.\lambda y. y \ ((\lambda z. z z) \ (\lambda z. z z))$$

redex

$$\lambda z. z z \ (\lambda z. z z)$$

is the leftmost innermost redex, and the redex

$$\lambda x.\lambda y. y \ ((\lambda z. z z) \ (\lambda z. z z))$$

is the leftmost outermost redex.
We saw that AOR may fail to terminate on expressions on which NOR terminates with a normal form.

On the other hand, AOR is more efficient. Consider for example

\[(\lambda x.g \ x \ x) \ ((\lambda y.f \ y) \ 3)\]
\[\rightarrow_\beta (\lambda x.g \ x \ x) \ (f \ 3)\]
\[\rightarrow_\beta g \ (f \ 3) \ (f \ 3)\]

whereas NOR requires one more step:

\[(\lambda x.g \ x \ x) \ ((\lambda y.f \ y) \ 3)\]
\[\rightarrow_\beta g \ ((\lambda y.f \ y) \ 3) \ ((\lambda y.f \ y) \ 3)\]
\[\rightarrow_\beta g \ (f \ 3) \ ((\lambda y.f \ y) \ 3)\]
\[\rightarrow_\beta g \ (f \ 3) \ (f \ 3)\]

The call-by-name argument passing creates copies of expressions containing redexes.

In functional languages, AOR implements *eager* evaluation and NOR implements *lazy* evaluation.
Suppose that AOR and NOR both terminate on a given expression. The question is: does the application of AOR or NOR on this expression result in the same normal form?

First, we have to prove that no choice of beta redex to reduce will result in a dead end.

A relation $\rightarrow$ satisfies the diamond property if for all terms $L, M_1, M_2$ such that $L \rightarrow M_1$ and $L \rightarrow M_2$ there exists a term $N$ such that $M_1 \rightarrow N$ and $M_2 \rightarrow N$.

**Theorem**

If a relation satisfies the diamond property, so does the transitive closure.
The $\rightarrow_\beta$ relation does not possess the diamond property. However, we can prove that $\not\rightarrow_\beta$ does in fact has the diamond property (see textbook).

**Theorem (Church Rosser)**

If $L \equiv_\beta M$, then there is an $N$ such that

$$L \not\rightarrow_\beta N \text{ and } M \not\rightarrow_\beta N$$

As a result of this, we can prove the following theorem stating that there is no “dead end” that prohibits reaching the normal form of a lambda expression.

**Theorem**

If $L \equiv_\beta N$ and $N$ is in normal form, then

$$L \not\rightarrow_\beta N$$

Now, we need another important property stating that normal forms are unique.

**Theorem**

A lambda expression can have at most one normal form.

Concluding the above, we can say that if an expression can be reduced in two different ways to normal forms then these normal forms are the same up to alphabetic equivalence.

**Theorem (Standardization Theorem)**

If a lambda expression $L$ has a normal form then reducing the left-most outermost redex at each stage in the reduction of $L$ guarantees reaching that normal form.
The \textit{eta reduction} relation $\eta$ encapsulates the idea that the two expressions
\[ \lambda x. L \, x \]
and
\[ L \]
denote the same function, provided that $x \notin \text{FV}[L]$ because
\[ (\lambda x. L \, x) \, M \rightarrow_{\beta} L \, M \]
for any lambda expression $M$.

This principle is called \textit{functional extensibility}. The associated eta contraction is defined as
\[ \lambda V. L \, V \eta \rightarrow L \]
for lambda expression $L$ and variable $V$ such that $V \notin \text{FV}[L]$.

For example
\[ (\lambda x. \sin x) \, 1 \rightarrow_{\eta} \sin 1 \]
We saw that beta reduction of a lambda expression may require variable renaming to avoid the variable clash problem.

Another solution to avoid the variable clash problem altogether is to use a restricted form of beta reduction resulting in a restricted normal form called \textit{weak head-normal form} (WHNF). An expression \( L \) is in WHNF if

- \( L \) is a function symbol.
- \( L \) is an expression of the form \( \lambda V . M \) for any \( M \).
- \( L \) is of the form \( F L_1 L_2 \ldots L_n \) for any function symbol \( F \) of arity \( k > n \).

For example

\[
\lambda x . (\lambda y . \lambda x . + y x) \ x
\]

is in WHNF (\((\lambda y . \lambda x . + y x) \ x\) will not be reduced). Only when this expression is applied to an argument, evaluation will take place. For example:

\[
(\lambda x . (\lambda y . \lambda x . + y x) \ x) \ 4
\]

\[
\rightarrow_\beta (\lambda y . \lambda x . + y x) \ 4
\]

\[
\rightarrow_\beta \lambda x . + 4 \ x
\]

Renaming is not necessary, because the problematic free variables are replaced by argument values.

Note that in pure lambda expression syntax the infix plus is written as a prefix operator (e.g. \( + \ 1 \ 2 = 3 \)).
We saw that NOR reduces the leftmost outermost beta redex first. This reduction order when combined with WHNF is comparable to the call-by-name calling mechanism.

For example, consider

$$(\lambda x. + x x) \ast 1 2$$

beta reduction of this expression yields

$$+ (\ast 1 2) \ast 1 2$$

and reduction of this expression requires the evaluation of ( $\ast 1 2$ ) twice. Hence, NOR can be potentially inefficient.

Sharing is a technique that avoids re-evaluation of arguments. Given an expression of the form

$$(\lambda x \ldots . x \ldots x \ldots) \ L$$

the beta reduced expression is written as

$$\ldots x \ldots x \ldots \quad \text{where } x \text{ is } L$$

Whenever any one of the $x$’s in the expression is reduced, all of the $x$’s will be bound to the resulting value. If none of the $x$’s are reduced, $L$ is not reduced. For example

$$(\lambda x. + x x) \ast 1 2$$

$\rightarrow_\beta + x x \quad \text{where } x \text{ is } ( \ast 1 2)$$

$\rightarrow_\beta + 2 x \quad \text{where } x \text{ is } 2$$

$\rightarrow_\beta + 2 2$$

$\rightarrow_\beta 4$$
We can say that

\textit{Eager evaluation} = AOR to WHNF

and

\textit{Lazy evaluation} = NOR to WHNF

+ sharing

+ lazy constructors

\textit{Lazy constructors} are functions forming data structures, for example lists, tuples, trees, etc. The arguments of constructors are not evaluated neither are the constructors applied to the arguments. One of the most basic constructors is the list constructor taking two arguments: the \textit{head} and \textit{tail} to form a list.

For example, in our demo lazy language \textit{Monica}, the list constructor is dot (.) and the empty list is []

\begin{verbatim}
> 1.[].
[1] :: [num]
\end{verbatim}

Monica shows the expression type to the right. Note that Monica requires input to be terminated with a period. Therefore, it is more convenient to write lists using [ and ]

\begin{verbatim}
> [1,2,3].
[1, 2, 3] :: [num]
\end{verbatim}

In Monica the tuple constructor is comma (,)

\begin{verbatim}
> 1+2,7,"abc".
(3, 7, "abc") :: (num, num, string)
> [5], 6.
([5], 6) :: ([num], num)
\end{verbatim}
You can use Monica to experiment with lambda calculus, so you can get a more practical feeling for it.

A lambda abstraction is written in Monica

\[ \lambda x. \text{body} \equiv x \to \text{body} \]

Monica adopts WHNF, so \text{body} is not evaluated.

A lambda application is written in Monica

\[ f \ arg_1 \ arg_2 \ldots \ arg_n \equiv f(arg_1, arg_2, \ldots, arg_n) \]

where \( f \) is an identifier. If \( f \) is a lambda expression, applications have to be written with colons

\[ f \ arg_1 \ arg_2 \ldots \ arg_n \equiv f:arg_1:arg_2:\ldots:arg_n \]

For example

```plaintext
> sin(1).
0.841471 :: num
> (x->x+1):2.
3 :: num
> (x->y->x+y):1:2.
3 :: num
> (x->y->x+y):1.
$0->1+$0 :: num->num
> x->x.
$0->$0 :: $0->$0
```

Monica renames variables into \$i where \( i \) is an integer \( \geq 0 \).
How do we express recursion in lambda calculus? Since functions are anonymous, we cannot refer to the function itself within the body of the function!

Consider for example

\[ \text{fac}(n) := \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{fac}(n - 1) & \text{else} \end{cases} \]

which can be written as

\[ \text{fac} := \lambda n. \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{fac}(n - 1) & \text{else} \end{cases} \]

By abstracting over the function body, we obtain

\[ \lambda f. \lambda n. \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n - 1) & \text{else} \end{cases} \]

The only thing that we have to do now is to bind the \( f \) to the lambda expression itself. This is accomplished with a special function \( Y \) called the (least) fixed point combinator:

\[ Y(\lambda f. \lambda n. \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n - 1) & \text{else} \end{cases}) \]

A graphical depiction of the result of the \( Y \) combinator is shown below.

\[ Y(\lambda f. \lambda n. \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n - 1) & \text{else} \end{cases}) \]
The fixpoint combinator $Y$ satisfies
\[ Y(f) = f(Yf) \]
When we apply $Y$ to
\[ F = \lambda f . \lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \ast f(n - 1) \]
we obtain
\[
YF \\
= Y(\lambda f . \lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \ast f(n - 1)) \\
\rightarrow \lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \ast (YF)(n - 1) \\
= \lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \ast \\
(\lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \ast (YF)(n - 1)) \\
(n - 1) \\
\rightarrow \vdots
\]
The question is, can we find a lambda expression for $Y$?
Recall that $(\lambda x . x x) (\lambda x . x x)$ is a lambda expression that replicates itself. The lambda expression for the $Y$ combinator is based on this principle:
\[ Y = \lambda h . (\lambda x . h (x x)) (\lambda x . h (x x)) \]
Let’s see if it works:
\[
Yf \\
= (\lambda h . (\lambda x . h (x x)) (\lambda x . h (x x))) f \\
\rightarrow (\lambda x . f (x x)) (\lambda x . f (x x)) \\
\rightarrow f ((\lambda x . f (x x)) (\lambda x . f (x x))) \\
= f (Yf)
\]
A fixed point (or fixpoint for short) of a function $f$ is an expression $e$ such that $f\ e = e$. A function may have none or more than one fixpoint.

Consider for example

$$f = \lambda x. x^2 - 6$$

Function $f$ has fixpoints $-2$ and $3$.

The fixpoint of $f$ is a constant. We are generally interested in fixpoints that are functions. For example, we are looking for a function that satisfies

$$f(0) = 1; \quad f(n) = n \times f(n - 1)$$

which we can write as

$$f = \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \times f \ (n - 1)$$

This equation is cast into an abstraction

$$F = \lambda f. \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \times f \ (n - 1)$$

The solution we seek is the fixpoint of $F$.

The fixpoint is computed by the sequence

$$F_0 = F(\Omega) = \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \times \Omega \ (n - 1)$$

$$F_1 = F(F_0) = \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \times F_0 \ (n - 1)$$

$$: = :$$

$$F_i = F(F_{i-1}) = \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \times F_{i-1} \ (n - 1)$$

where $\Omega$ can be any function.
The idea is to make as many copies of $F$ as possible in order to solve the problem. For example, $F_2$ works for $n = 0, 1, 2$:

$$F_2 \ 2 \ = \ \textbf{if} \ (2 = 0) \ \textbf{then} \ 1 \ \textbf{else} \ 2 \ * \ F_1 \ (2 - 1)$$
$$= \ 2 \ * \ F_1 \ 1$$
$$= \ 2 \ * \ (\textbf{if} \ (1 = 0) \ \textbf{then} \ 1 \ \textbf{else} \ 1 \ * \ F_0 \ (1 - 1))$$
$$= \ 2 \ * \ (1 \ * \ F_0 \ 0)$$
$$= \ 2 \ * \ (1 \ * \ (\textbf{if} \ (0 = 0) \ \textbf{then} \ 1 \ \textbf{else} \ 1 \ * \ \Omega \ (0 - 1)))$$
$$= \ 2 \ * \ (1 \ * \ 1)$$
$$= \ 2$$

This is what the $Y$ combinator will do. It creates as many copies as necessary to compute the fixed point if it exists, i.e. a function satisfying the recursive equation. This function is the least fixed point.
There is possibly more than one fixpoint of a function. The $Y$ combinator returns the least fixed point of a function. This assumes an ordering on functions, defined by

$$f_1 \sqsubseteq f_2 \text{ if } f_1(x) = f_2(x) \text{ or } f_1(x) = \textit{undefined} \quad \forall x \in \text{Dom}(f_2)$$

Consider all the functions $f$ that satisfy

$$f(0) = 1; \quad f(n) = 2 \times f(f(n - 1))$$

The example functions below satisfy the equation:

1. $g(n) = \begin{cases} 1 & \text{if } n = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$

2. $h(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ 4 & \text{if } n = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$

3. $j(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 + 3i, i = 0, 1, \ldots \\ 2 & \text{if } n = 2 + 3i, i = 0, 1, \ldots \\ 4 & \text{if } n = 3 + 3i, i = 0, 1, \ldots \end{cases}$

We have $g \sqsubseteq h \sqsubseteq j$. We can write this problem as

$$F = \lambda f . \lambda n . \text{if } (n = 0) \text{ then } 1 \text{ else } 2 \times (f (f (n - 1)))$$

and apply the $Y$ combinator to find a solution: $YF = g$. 
Let us now consider the expression of Boolean values, conditional expressions, numbers, lists, etc. Can these be expressed in pure lambda calculus?

The Boolean values TRUE and FALSE are implemented as

\[ \text{TRUE} := \lambda x.\lambda y.x \]
\[ \text{FALSE} := \lambda x.\lambda y.y \]

That is, they select the first or second argument, respectively.

A conditional expression is implemented as

\[ \text{if } P \text{ then } T \text{ else } E := P T E \]

This works because \( P = \text{TRUE} \) selects \( T \) and \( P = \text{FALSE} \) selects \( E \), e.g:

\[
\text{TRUE} \ T \ E \\
= (\lambda x.\lambda y.x) \ T \ E \\
\rightarrow_\beta (\lambda y.T) \ E \\
\rightarrow_\beta T
\]

Furthermore, we can define

\[ \text{AND} := \lambda x.\lambda y.\lambda z.x \ y \ \text{FALSE} \]
\[ \text{OR} := \lambda x.\lambda y.\lambda z.x \ \text{TRUE} \ y \]
\[ \text{NOT} := \lambda x.\lambda y.\lambda z.x \ z \ y \]
Function CONS builds a list from an element and an old list:

\[
\text{CONS} := \lambda h . \lambda t . \lambda s . s \ h \ t
\]

Note: In the textbook the macro \(\text{Pair}(h, t)\) is used:

\[
\text{Pair}(h, t) := \text{CONS } h \ t = \lambda s . s \ h \ t
\]

The role of \(s\) is to serve as a selector function to get the head \(h\) or tail \(t\) of the list.

As head and tail selector functions \(s\) for the CONS list, the TRUE and FALSE functions can be used:

\[
\text{HD} := \lambda L . L \ \text{TRUE}
\]

\[
\text{TL} := \lambda L . L \ \text{FALSE}
\]

where \(L\) is assumed to be a CONS list. For example

\[
\text{HD } (\text{CONS } a \ b)
\]

\[
= (\lambda L . L \ \text{TRUE}) \ ((\lambda h . \lambda t . \lambda s . s \ h \ t) \ a \ b)
\]

\[
\rightarrow_\beta (\lambda h . \lambda t . \lambda s . s \ h \ t) \ a \ b \ \text{TRUE}
\]

\[
\rightarrow_\beta (\lambda t . \lambda s . s \ a \ t) \ b \ \text{TRUE}
\]

\[
\rightarrow_\beta (\lambda s . s \ a \ b) \ \text{TRUE}
\]

\[
\rightarrow_\beta \ \text{TRUE} \ a \ b
\]

\[
= (\lambda x . \lambda y . x) \ a \ b
\]

\[
\rightarrow_\beta (\lambda y . a) \ b
\]

\[
\rightarrow_\beta a
\]

The constant NIL denotes the empty list, so for example

\[
\text{CONS } 1 \ \text{NIL} = \lambda s . s \ 1 \ \text{NIL}
\]

which is the singleton list containing element 1.
To complete the set of list functions, we have to define the *is empty* or *null* function IE with the requirement that

\[ \text{IE (CONS } a \ b) \rightarrow \text{FALSE} \]

and

\[ \text{IE NIL } \rightarrow \text{TRUE} \]

The following definition will do the trick:

\[ \text{IE} := \lambda c. c (\lambda h. \lambda t. \text{FALSE}) \]

To see why this works, consider

\[ \text{IE (CONS } a \ b) \]

\[ = (\lambda c. c (\lambda h. \lambda t. \text{FALSE})) (\text{CONS } a \ b) \]

\[ \rightarrow_\beta \ \text{CONS } a \ b (\lambda h. \lambda t. \text{FALSE}) \]

\[ = (\lambda h. \lambda t. \lambda s. s \ h \ t) \ a \ b (\lambda h. \lambda t. \text{FALSE}) \]

\[ \rightarrow_\beta (\lambda s. s \ a \ b) \ (\lambda h. \lambda t. \text{FALSE}) \]

\[ \rightarrow_\beta (\lambda h. \lambda t. \text{FALSE}) \ a \ b \]

\[ \rightarrow_\beta \ \text{FALSE} \]

Since we require that \( \text{IE NIL } = \text{TRUE} \), we define

\[ \text{NIL} := \lambda x. \text{TRUE} \]

This works:

\[ \text{IE NIL} \]

\[ = (\lambda c. c (\lambda h. \lambda t. \text{FALSE})) \ \text{NIL} \]

\[ \rightarrow_\beta \ \text{NIL} (\lambda h. \lambda t. \text{FALSE}) \]

\[ = (\lambda x. \text{TRUE}) (\lambda h. \lambda t. \text{FALSE}) \]

\[ \rightarrow_\beta \ \text{TRUE} \]
The natural numbers in pure lambda calculus are based on lists. We will use the representation explained in the extra handouts. The textbook gives a closely related alternative representation.

A number \( n \) is viewed as an \( n \)-element list of objects of arbitrary value (c.f. tally notation).

- Base \( n = 0 \) is represented by NIL.
- Successor is defined by \( \text{SUCC} := \lambda n . \text{CONS} \ \text{anything} \ n \).

Here, \( \text{anything} \) can be \( \text{any} \) lambda expression.

For example

\[
\text{SUCC NIL} = \text{CONS} \ \text{anything} \ \text{NIL}
\]

\[
\text{SUCC (SUCC NIL)} = \text{CONS} \ \text{anything} \ (\text{CONS} \ \text{anything} \ \text{NIL})
\]

etc.

Clearly, the \textit{is zero} function is just IE, because 0 is represented by the empty list.

The predecessor function is defined by

\[
\text{PRED} := \text{TL}
\]

This works because the tail of a list represents the previous natural number.
In lambda calculus, substitution requires variable renaming when variable names conflict. Sometimes, the use of variables seems to be unnatural and not essential for describing function properties. For example

\[ x^2 \geq 0 \quad \forall x \in \mathbb{R} \]

states a property of squaring, not about the variable \( x \).

Combinators eliminate the need for variables. Combinatory calculus is constructed out of the fusion combinator \( S \), the constant combinator \( K \), and identity combinator \( I \). The only primitive operation is application written as \( fa \) (\( f \) applied to \( a \)).

\( S \) and \( K \) are defined by the contractions

\[
S \ x \ y \ z \ >_C \ x \ z \ (y \ z) \\
K \ x \ y \ >_C \ x
\]

The \( I \) combinator can be expressed in terms of \( S \) and \( K \):

\[
I := S \ K \ K
\]

since we have the reduction

\[
I \ x = S \ K \ K \ x \rightarrow_C K \ x \ (K \ x) \rightarrow_C x
\]
A **combinator term** is an expression $X$ given by

\[
\begin{align*}
V & \to x \mid y \mid z \mid u \mid v \mid \ldots \\
F & \to a \mid b \mid c \mid f \mid g \mid \ldots \\
X & \to V \mid F \mid S \mid K \mid I \mid X X \mid (X)
\end{align*}
\]

Note: application $(X X)$ is left associative.

In contrast to lambda calculus, there is no abstraction operator in combinatory calculus. Hence, a variable that appears in a combinator term is always a free variable. By definition, a combinator is a combinator term that is closed.

Example combinator terms are

\[
\begin{align*}
\text{K (I } x) & \quad FV = \{x\} \\
f y & \quad FV = \{y\} \\
x y z & \quad FV = \{x, y, z\} \\
x (y z) & \quad FV = \{x, y, z\} \\
x (K y z) & \quad FV = \{x, y, z\} \\
\text{S K I} & \quad \text{is closed: S K I is a combinator}
\end{align*}
\]
The mapping $M[X] = L$ from combinator terms $X$ to lambda expressions $L$ is defined by

$$
M[x] := x \\
M[f] := f \\
M[I] := \lambda x . x \\
M[K] := \lambda x . \lambda y . x \\
M[S] := \lambda x . \lambda y . \lambda z . x z (y z) \\
M[X_1 X_2] := M[X_1] M[X_2]
$$

This mapping is directly based on the $>^C$ contraction relation for $S$ and $K$. Note that the mapping describes the semantics of combinatory calculus in terms of lambda calculus. For example

$$
= (\lambda x . \lambda y . \lambda z . x z (y z)) (\lambda v . v) (\lambda v . v) (M[S] M[I] M[I]) \\
\rightarrow_\beta (\lambda y . \lambda z . (\lambda v . v) z (y z)) (\lambda v . v) (M[S] M[I] M[I]) \\
\rightarrow_\beta (\lambda z . (\lambda v . v) z ((\lambda v . v) z)) (M[S] M[I] M[I]) \\
\rightarrow_\beta (\lambda z . z z) (M[S] M[I] M[I]) \\
= (\lambda z . z z) (\lambda z . z z)
$$
By applying combinator reduction we find

\[
\begin{align*}
    \text{SII (SII)} \\
    \rightarrow_C \text{ I (SII) (I (SII))} \\
    \rightarrow_C \text{ SII (I (SII))} \\
    \rightarrow_C \text{ SII (SII)}
\end{align*}
\]

In general, combinator reduction of a combinator term \( X \) can be expressed in terms of beta reduction on lambda expression \( L = \mathcal{M}[X] \). This gives rise to the following conceptual diagram

\[
\begin{array}{ccc}
    X & \rightarrow_C & X' \\
    L = \mathcal{M}[X] & \downarrow & L' \equiv_{\alpha} \mathcal{M}[X'] \\
    L & \rightarrow & L' \\
    & \downarrow & \downarrow \\
    & \rightarrow_{\beta} & \\
\end{array}
\]
To define a mapping from lambda expressions to combinator terms, we need to translate abstractions. To this end, the *bracket abstraction* algorithm will be used. The idea is to successively abstract away the variables from subexpressions.

Bracket abstraction of a variable $x$ is defined by

$$\begin{align*}
[x]y & := \begin{cases} I & \text{if } y = x \\ K y & \text{otherwise} \end{cases} \\
[x]f & := K f \\
[x]S & := K S \\
[x]K & := K K \\
[x]I & := K I \\
[x](L_1 L_2) & := S ([x]L_1) ([x]L_2)
\end{align*}$$

Bracket abstraction has the following property

$$( [x]L ) x \xrightarrow{\ast_C} L$$

That is, if the result of abstracting $x$ from lambda expression $L$ is applied back to $x$, this result reduces to $L$. Example:

$$\begin{align*}
[x]+ x 1 \\
= S ([x]+ x) ([x]1) \\
= S (S (K+) I) (K 1)
\end{align*}$$

Application of $S (S (K+) I) (K 1)$ to $x$ yields

$$\begin{align*}
S (S (K+) I) (K 1) x \\
\xrightarrow{C} S (K+) I x (K 1 x) \\
\xrightarrow{C} K + x (I x) (K 1 x) \\
\xrightarrow{\ast_C} + x 1
\end{align*}$$
With bracket abstraction the algorithm for translating lambda expres-
sions $L$ into combinator terms $X$ denoted $X = \mathcal{N}[L]$ is defined by

\[
\begin{align*}
\mathcal{N}[x] & := x \\
\mathcal{N}[f] & := f \\
\mathcal{N}[L_1 L_2] & := \mathcal{N}[L_1]\mathcal{N}[L_2] \\
\mathcal{N}[\lambda x . L] & := [x]\mathcal{N}[L]
\end{align*}
\]

Example
\[
\begin{align*}
\mathcal{N}[\lambda x . + x 1] & = [x]\mathcal{N}[+ x 1] \\
& = [x](+ x 1) \\
& = S (S (K +) I) (K 1) \quad \text{see prev. transparency}
\end{align*}
\]

**Theorem**
For all lambda expressions $L$ we have $\mathcal{M}[\mathcal{N}[L]] \equiv L$ where the equivalence is alpha-beta-eta equivalence.

Example
\[
\begin{align*}
\mathcal{M}[\mathcal{N}[\lambda x . + x 1]] & = \mathcal{M}[S (S (K +) I) (K 1)] \\
& = \mathcal{M}[S] (\mathcal{M}[S] (\mathcal{M}[K] \mathcal{M}[+]) \mathcal{M}[I]) (\mathcal{M}[K] \mathcal{M}[1]) \\
& = (\lambda x . \lambda y . \lambda z . x z (y z)) \\
& \quad (((\lambda x . \lambda y . \lambda z . x z (y z)) ((\lambda x . \lambda y . x) +) (\lambda x . x)) \\
& \quad (((\lambda x . \lambda y . x) 1) \\
& \quad \xrightarrow{\beta} \lambda z . + z 1
\end{align*}
\]
A disadvantage of the bracket abstraction algorithm based on $S$, $K$, and $I$ is that the resulting combinator term can be exponentially longer than the initial lambda expression.

Consider for example

$$N[\lambda x . \lambda y . + x y]$$

$$= [x]N[\lambda y . + x y]$$
$$= [x]([y]N[+ x y])$$
$$= [x]([y](+ x y))$$
$$= [x](S ([y](+ x)) ([y]y))$$
$$= [x](S (S ([y]+) ([y]x)) ([y]y))$$
$$= [x](S (S (K+) (Kx)) I)$$
$$= S ([x](S (S (K+) (Kx)))) ([x]I)$$
$$= S (S ([x]S) ([x](S (K+) (Kx)))) (KI)$$
$$= S (S (KS) ([x](S (K+) (Kx)))) (KI)$$
$$= S (S (KS) S (S ([x]S) ([x](K+))) ([x](K x))) (KI)$$
$$= S (S (KS) S (S (KS) (S ([x]K) ([x]+))) ([x](K x))) (KI)$$
$$= S (S (KS) S (S (KS) (S (KK) (K+))) ([x](K x))) (KI)$$
$$= S (S (KS) S (S (KS) (S (KK) (K+))) (S ([x]K) ([x]x))) (KI)$$
$$= S (S (KS) S (S (KS) (S (KK) (K+))) S (KK) I) (KI)$$

which seems to be an extraordinary complicated way of adding two arguments.
Curry used two extra combinators $B$ and $C$ with the following meaning:

$$B \ x \ y \ z \ >_C \ x \ (y \ z)$$
$$C \ x \ y \ z \ >_C \ x \ z \ y$$

Curry’s bracket algorithm is slightly different:

$$[x]y := \begin{cases} I & \text{if } y = x \\ K_y & \text{otherwise} \end{cases}$$

$$[x]f := Kf$$ for any function symbol $f$

$$[x](L \ x) := L$$ if $x \not\in FV[L]$

$$[x](L_1 \ L_2) := K \ (L_1 \ L_2)$$ if $x \not\in FV[L_1] \land x \not\in FV[L_2]$

$$[x](L_1 \ L_2) := C \ ([x]L_1 \ L_2)$$ if $x \in FV[L_1] \land x \not\in FV[L_2]$

$$[x](L_1 \ L_2) := B \ L_1 \ ([x]L_2)$$ if $x \not\in FV[L_1] \land x \in FV[L_2]$

$$[x](L_1 \ L_2) := S \ ([x]L_1) \ ([x]L_2)$$ if $x \in FV[L_1] \land x \in FV[L_2]$

This bracket algorithm generates combinator terms that are at worst cubic in the length of the original lambda expression. Reconsider the previous example:

$$\mathcal{N}[\lambda \, x . \, \lambda \, y \, . \, + \, x \, y]$$

$$= [x] \mathcal{N}[\lambda \, y \, . \, + \, x \, y]$$

$$= [x]([y](+ \ x \ y))$$

$$= [x](+ \ x)$$

$$= +$$
Combinators can be represented as trees. Correspondingly, combinator reduction amounts to *graph reduction*.

Graph reduction on combinator terms can be applied independently, therefore parallel processing is possible with a graph reduction machine.
Textbook exercises 7.6, 7.17, 7.20, 7.21. Give the complete derivations of the following exercises.

1. Write down the normal form (if it exists) and weak-head normal form of each of the following lambda expressions showing the reduction steps required to reach them with NOR. Be careful to avoid the name-clash problem.

(i) \((\lambda x. (\lambda y. x) 1) (\lambda z. z)\)
Answer: 1 (WHNF and NF)

(ii) \((\lambda x. \lambda y. x) (\lambda z. z)\)
Answer: 2 (WHNF and NF)

(iii) \((\lambda x. \lambda y. (\lambda z. z) x) (+ y 1)\)
Answer: \(\lambda v. (\lambda z. z) (+ y 1)\) (WHNF) \(\lambda v. + y 1\) (NF)

(iv) \((\lambda x. \lambda y. x (\lambda z. y z)) (\lambda x. (\lambda y. y) x)\)
Answer: \(\lambda y. (\lambda x. \lambda y. y) 8 (\lambda x. (\lambda y. y) x) (\lambda z. y z)\) (WHNF) \(\lambda y. \lambda z. y z\) (NF)

(v) \((\lambda h. (\lambda x. h (x x)) (\lambda x. h (x x)))\)
Answer: no (WH)NF: does not terminate

2. Show that OR TRUE FALSE \(\equiv_{\alpha\beta}\) TRUE

using the pure lambda calculus definitions of TRUE, FALSE, and OR. \(\text{Hint:}\) Reduce OR TRUE FALSE to normal form. This form should be \(\alpha\)-equivalent to the lambda expression for TRUE

3. Apply Curry’s optimized bracket abstraction algorithm to compute \(X = \mathcal{N}[L]\) where \(L\) is \(\lambda x. \lambda y. x\)

Answer: \(\mathcal{N}[\lambda x. \lambda y. x] = [x]\mathcal{N}[\lambda y. x] = [x][([y]\mathcal{N}[x])] = [x][([y][x])] = [x](K) x = K\)

\(\lambda x. \lambda y. x\) \((x y)\)

Answer: \(\mathcal{N}[\lambda x. \lambda y. x (x y)] = [x]\mathcal{N}[\lambda y. x (x y)] = [x][([y]\mathcal{N}[x (x y)])] = [x][([y][x (x y)])] = [x](B) x ([y][x (x y)])\)
“We want the meaning of languages to be complete, consistent, precise, unambiguous, concise, understandable, and useful.”

In contrast to syntax, the meaning of a programming language is often described informally. This is fine up to some point.

Denotational semantics is an example of formal semantics: a means of describing programming languages more precisely using formal methods.

Edsger Dijkstra is known to be the strongest advocate of formal methods in computer science as a means of program verification.

Many discussions have taken place between supporters and opposers of formal methods in computer science, and discussions are still ongoing.

Formal semantics are useful, but it remains a challenge to apply formal methods in routine software development.
When are formal semantics useful?

- **Standardization of programming languages**
  Standards must be precise in minute detail

- **Language design**
  If the formal semantics of a programming construct are complex, then the language construct might be difficult to use

- **Reference for implementors**
  A formal description should prevent incompatible implementations of the same language

- **Automatic implementation of compilers**
  If the formal description is complete, the language implementation can be automated

- **Proof of program correctness**
  Some critical systems require formal verification of the system software
The primary techniques used for describing the formal semantics of programming languages are

- **Operational semantics**
  A semantic description of a programming construct is given in terms of instructions of a primitive (abstract) machine

- **Denotational semantics**
  This chapter: functions map programs to the abstract mathematical values they denote

- **Axiomatic semantics**
  Chapter 9: the action of a program construct is defined by the logical properties of the state of the machine before and after execution of the construct

- **Algebraic semantics**
  Programs implement functions that can be expressed by equations between terms

- **Structured operational semantics**
  or **natural semantics**
  Programs are given meaning by derivation rules that describe the evaluation of the constructs (see exercise 4.15).
Operational semantics depends on algorithms, not mathematics. Primitive instructions are used to represent programming constructs.

As an example, the operational semantics in the description of the for construct in C might be:

\[ M[\textbf{for} \ (e1; \ e2; \ e3) \ { \ s \ }] \]

\[ = \ M[e1] ; \]

\[ \text{loop: if } M[e2 == 0] \ \text{ goto out} \]

\[ M[s] \]

\[ M[e3] \]

\[ \text{ goto loop} \]

\[ \text{out: ...} \]

where \( M \) is a mapping from programs to the primitive instructions of an abstract machine.

The following list of instructions is sufficient to describe any programming construct:

\[ \text{ident := ident} \]

\[ \text{ident := const} \]

\[ \text{ident := ident } \otimes \text{ ident} \]

\[ \text{ident := } \oplus \text{ ident} \]

\[ \text{goto label} \]

\[ \text{if ident goto label} \]

where \( \otimes \in \{+, -, *, /, \text{div}, \text{mod}, \ll, \gg, \land, \lor\} \) and \( \oplus \in \{-, +, -\} \).
The drawback of operational semantics is the weakness of the approach. For example, the programming language of the abstract machine may not have been defined in detail. To define the behavior of the machine we need (operational) semantics! This possibly leads to circular definitions.

The denotational approach associates mathematical denotations to syntactical parts of a language. The denotations are (presumably) familiar mathematical objects. Functions are used to map syntactic constructs to the mathematical objects.

For example, to define meaning to the string of binary digits \(01011011\) we can define

\[
D[01011011] := 91
\]

syntactic construct \(\mapsto\) mathematical object

The mapping \(D\) uniquely defines the semantics of the string \(01011011\) by the assumption that 91 is a familiar mathematical number.
Ch. 8  Denotational Semantics: Procedure

To set up the denotational semantics of a language, we proceed in the following four steps:

1. *Give the syntactic categories*

2. *Define the (abstract) syntax of the language*
   Write down the syntax of the language under consideration. The syntax can be either concrete or abstract.

3. *Give the value domains*
   Give the familiar mathematical objects that serve as the target of the definition.

4. *Define functions that map each syntactic object to some semantic value*
   Write the functions that convey the meaning of the syntactic constructs of the language by mapping the constructs to the mathematical objects. The definition is inductive on the structure of the language.
The language of decimal numerals.

1. **Syntactic categories**
   
   \[ D \in \text{Digits} \quad \text{the decimal digits} \]
   
   \[ N \in \text{Num} \quad \text{decimal numerals} \]

2. **Syntax**
   
   \[ D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]
   
   \[ N \rightarrow D \mid N \, D \]

3. **Value domains**
   
   \[ \text{Nat} = \{0, 1, 2, \ldots\} \]

4. **Semantic functions**
   
   \[ \mathcal{D} : \text{Digits} \rightarrow \text{Nat} \]
   
   \[ \mathcal{M} : \text{Num} \rightarrow \text{Nat} \]

   inductively defined by

   \[ \mathcal{D}[0] = 0 \]
   
   \[ \mathcal{D}[1] = 1 \]
   
   \[ \vdots \]
   
   \[ \mathcal{D}[9] = 9 \]
   
   \[ \mathcal{M}[D] = \mathcal{D}[D] \]
   
   \[ \mathcal{M}[N \, D] = 10 \times \mathcal{M}[N] + \mathcal{M}[D] \]
The language of regular expressions.

1. **Syntactic categories**
   
   \[ A \in \text{Alpha} \quad \text{the alphabet (includes } \epsilon \text{)} \]
   
   \[ R \in \text{RE} \quad \text{regular expressions} \]

2. **Syntax**
   
   \[ R \rightarrow A | \emptyset | ( R | R ) | ( R R ) | R^* \]

3. **Value domains**
   
   \[ \text{Lang} \quad \text{languages (sets of strings over } \text{Alpha} \text{)} \]

4. **Semantic functions**
   
   \[ \mathcal{A} : \text{Alpha} \rightarrow \text{Lang} \]
   
   \[ \mathcal{D} : \text{RE} \rightarrow \text{Lang} \]

   inductively defined by

   \[ \mathcal{A}[A] = \{ A \} \]
   
   \[ \mathcal{D}[A] = \mathcal{A}[A] \]
   
   \[ \mathcal{D}[\emptyset] = \emptyset \]
   
   \[ \mathcal{D}[R_1 | R_2] = \mathcal{D}[R_1] \cup \mathcal{D}[R_2] \]
   
   \[ \mathcal{D}[R_1 R_2] = \{ x \cdot y \mid x \in \mathcal{D}[R_1] \land y \in \mathcal{D}[R_2] \} \]
   
   \[ \mathcal{D}[R^*] = \bigcup_{i=0}^{\infty} \mathcal{D}[R]^i \]

   where \( s^0 = \{ \epsilon \} \) and \( s^{(i+1)} = \{ x \cdot y \mid x \in s \land y \in s^i \} \) for set \( s \).
The language of propositional logic.

1. **Syntactic categories**

   \[ P \in \text{Prop} \quad \text{the propositions} \]
   \[ F \in \text{Form} \quad \text{formulas} \]

2. **Syntax**

   \[ F \rightarrow P \lor \neg F \lor F \Rightarrow F \]

3. **Value domains**

   \[ \text{Bool} = \{ \text{TRUE}, \text{FALSE} \} \quad \text{Boolean values} \]
   \[ \rho \in \text{Assign} = \text{Prop} \rightarrow \text{Bool} \quad \text{assignments} \]

4. **Semantic functions**

   \[ \mathcal{M} : \text{Form} \rightarrow \text{Assign} \rightarrow \text{Bool} \]

   inductively defined by

   \[ \mathcal{M}[P] \rho = \rho(P) \]
   \[ \mathcal{M}[\neg F] \rho = \neg \mathcal{M}[F] \rho \]
   \[ \mathcal{M}[F_1 \Rightarrow F_2] \rho = \neg \mathcal{M}[F_1] \rho \lor \mathcal{M}[F_2] \rho \]
The language of simple expressions.

1. **Syntactic categories**
   
   \[ I \in \text{Ident} \quad \text{identifiers} \]
   
   \[ E \in \text{Exp} \quad \text{expressions} \]

2. **Syntax**
   
   \[ E \rightarrow 0 \mid 1 \mid I \mid E + E \mid E - E \mid \text{let } I = E \text{ in } E \text{ end} \]

3. **Value domains**
   
   \[ \textbf{Int} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \} \quad \text{integers} \]
   
   \[ \rho \in \textbf{Env} = \text{Ident} \rightarrow \textbf{Int} \quad \text{environments} \]

4. **Semantic functions**
   
   \[ \mathcal{E} : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Int} \]
   
   inductively defined by
   
   \[ \mathcal{E}[0] \rho = 0 \]
   
   \[ \mathcal{E}[1] \rho = 1 \]
   
   \[ \mathcal{E}[I] \rho = \rho(I) \]
   
   \[ \mathcal{E}[E_1 + E_2] \rho = \mathcal{E}[E_1] \rho + \mathcal{E}[E_2] \rho \]
   
   \[ \mathcal{E}[E_1 - E_2] \rho = \mathcal{E}[E_1] \rho - \mathcal{E}[E_2] \rho \]
   
   \[ \mathcal{E}[\text{let } I = E_1 \text{ in } E_2 \text{ end}] \rho = \mathcal{E}[E_2] (\rho[I \mapsto \mathcal{E}[E_1] \rho]) \]
Language of simple expressions with error values.

1. **Syntactic categories**

   - \( I \in \text{Ident} \) identifiers
   - \( E \in \text{Exp} \) expressions
   - \( P \in \text{Prog} \) programs

2. **Syntax**

   \[
   E \rightarrow 0 \mid 1 \mid I \mid E + E \mid E / E
   \mid \text{let } I = E \text{ in } E \text{ end} \mid \text{if } E \text{ then } E \text{ else } E
   \]

   \[
   P \rightarrow \text{program}(I); \ E \text{ end}.
   \]

3. **Value domains**

   - \( \text{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \) integers
   - \( \rho \in \text{Env} = \text{Ident} \rightarrow (\text{Int} + \bot) \) environments

4. **Semantic functions**

   \[
   \mathcal{E} : \text{Exp} \rightarrow \text{Env} \rightarrow (\text{Int} + \text{error})
   \]

   \[
   \mathcal{P} : \text{Prog} \rightarrow \text{Int} \rightarrow (\text{Int} + \text{error})
   \]

   \[
   \mathcal{E}[0] \rho = 0
   \]

   \[
   \mathcal{E}[1] \rho = 1
   \]

   \[
   \mathcal{E}[I] \rho = \text{let } n = \rho(I)
   \text{ in if IsInt}(n) \text{ then } n \text{ else error}
   \text{ end}
   \]

   \[
   \mathcal{E}[E_1 + E_2] \rho
   \]
\[
= \text{let } n = \mathcal{E}[E_1] \rho; m = \mathcal{E}[E_2] \rho \\
\quad \text{in } \text{if IsInt}(n) \land \text{IsInt}(m) \text{ then } n + m \text{ else error} \\
\text{end}
\]

\[
\mathcal{E}[E_1/E_2] \rho \\
= \text{let } n = \mathcal{E}[E_1] \rho; m = \mathcal{E}[E_2] \rho \\
\quad \text{in } \text{if IsInt}(n) \land \text{IsInt}(m) \land m \neq 0 \text{ then } \lfloor n/m \rfloor \text{ else error} \\
\text{end}
\]

\[
\mathcal{E}[\text{if } E_1 \text{ then } E_2 \text{ else } E_3] \rho \\
= \text{let } n = \mathcal{E}[E_1] \rho \\
\quad \text{in } \text{if IsInt}(n) \\
\quad \text{then if } n \neq 0 \text{ then } \mathcal{E}[E_2] \rho \text{ else } \mathcal{E}[E_3] \rho \\
\quad \text{else error} \\
\text{end}
\]

\[
\mathcal{E}[\text{let } I = E_1 \text{ in } E_2 \text{ end}] \rho \\
= \text{let } n = \mathcal{E}[E_1] \rho \\
\quad \text{in } \text{if IsInt}(n) \text{ then } \mathcal{E}[E_2] \rho(I \mapsto n) \text{ else error} \\
\text{end}
\]

\[
\mathcal{P}[\text{program}(I); E \text{ end.}] \rho \\
= \text{let } J = \bot \\
\quad \text{in } \mathcal{E}[E] \rho(I \mapsto n) \\
\text{end}
\]
Language with state

1. Syntactic categories

   \[ I \in \text{Ident} \quad \text{identifiers} \]
   \[ E \in \text{Exp} \quad \text{expressions} \]
   \[ C \in \text{Com} \quad \text{commands} \]
   \[ P \in \text{Prog} \quad \text{programs} \]

2. Syntax

   \[
   E \rightarrow 0 \mid 1 \mid I \mid E + E \mid E - E \\
   \quad \mid \text{let } I = E \text{ in } E \text{ end} \mid \text{if } E \text{ then } E \text{ else } E \\
   C \rightarrow C ; C \mid \text{if } E \text{ then } C \text{ else } C \mid \text{while } E \text{ do } C \\
   P \rightarrow \text{program}(I) ; C \text{ end.}
   \]

3. Value domains

   \[
   \text{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \quad \text{integers} \\
   \sigma \in \text{States} \quad \text{states}
   \]

4. Semantic functions

   \[
   \mathcal{E} : \text{Exp} \rightarrow \text{States} \rightarrow \text{Int} \\
   \mathcal{C} : \text{Com} \rightarrow \text{States} \rightarrow \text{States} \\
   \mathcal{P} : \text{Prog} \rightarrow \text{Int} \rightarrow \text{Int}
   \]
Statement composition:
\[ C[C_1; C_2] \sigma = \text{let } \sigma' = C[C_1] \sigma \]
\[ \text{in } C[C_2] \sigma' \]
\[ \text{end} \]

or alternatively
\[ C[C_1; C_2] \sigma = C[C_2] (C[C_1] \sigma) \]

which can also be written
\[ C[C_1; C_2] = C[C_2] \circ C[C_1] \]

Conditional control flow:
\[ C[\text{if } E \text{ then } C_1 \text{ else } C_2] \sigma \]
\[ = \text{ if IsTrue}(E[C] \sigma) \text{ then } C[C_1] \sigma \text{ else } C[C_2] \sigma \]

Loop:
\[ C[\text{while } E \text{ do } C] \sigma \]
\[ = \text{ let } p(\sigma') = \text{if IsTrue}(E[C] \sigma') \text{ then } p(C[C] \sigma') \text{ else } \sigma' \]
\[ \text{in } p(\sigma) \]
\[ \text{end} \]

Program:
\[ P[\text{program}(I); C \text{ end.}] n \]
\[ = \text{ let } \sigma_f = C[C] \sigma_i \]
\[ \text{in } \text{Contents}(\sigma_f, \text{Address}(I)) \]
\[ \text{end} \]
Language with assignments

1. **Syntactic categories**

   \[ I \in \text{Ident} \quad \text{identifiers} \]
   \[ L \in \text{Lexp} \quad \text{l-values or references} \]
   \[ E \in \text{Exp} \quad \text{expressions} \]
   \[ C \in \text{Com} \quad \text{commands} \]
   \[ P \in \text{Prog} \quad \text{programs} \]

2. **Syntax**

   \[ L \rightarrow I \]
   \[ E \rightarrow 0 \mid 1 \mid I \mid !L \mid E + E \]
   \[ \mid \text{let } I = E \text{ in } E \text{ end} \mid \text{if } E \text{ then } E \text{ else } E \]
   \[ C \rightarrow C \mid C \mid \text{if } E \text{ then } C \text{ else } C \mid L := E \]
   \[ \mid \text{new } I = E \text{ in } C \text{ end} \]
   \[ P \rightarrow \text{program}(I); \ C \text{ end.} \]

3. **Value domains**

   \[ n \in \text{Int} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \} \quad \text{integers} \]
   \[ \alpha \in \text{Loc} \quad \text{locations} \]
   \[ \sigma \in \text{States} = \text{Loc} \rightarrow (\text{Int} + \text{unused}) \quad \text{states} \]
   \[ \rho \in \text{Env} = \text{Ident} \rightarrow (\text{Int} + \text{Loc} + \bot) \quad \text{environments} \]
4. Semantic functions

\[ \mathcal{L} : \text{Lexp} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{Loc} \]
\[ \mathcal{E} : \text{Exp} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{Int} \]
\[ \mathcal{C} : \text{Com} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{States} \]
\[ \mathcal{P} : \text{Prog} \rightarrow \text{Int} \rightarrow \text{Int} \]

L-values or references:

\[ \mathcal{L}[I] \rho \sigma = \rho(I) \]

Expressions:

\[ \mathcal{E}[0] \rho \sigma = 0 \]
\[ \mathcal{E}[1] \rho \sigma = 1 \]
\[ \mathcal{E}[I] \rho \sigma = \rho(I) \]
\[ \mathcal{E}[!L] \rho \sigma = \sigma(\mathcal{L}[L] \rho \sigma) \]
\[ \mathcal{E}[E_1 + E_2] \rho \sigma = \mathcal{E}[E_1] \rho \sigma + \mathcal{E}[E_2] \rho \sigma \]
\[ \mathcal{E}[\text{let } I = E_1 \text{ in } E_2 \text{ end}] \rho \sigma = \mathcal{E}[E_2] (\rho[I \mapsto \mathcal{E}[E_1] \rho \sigma]) \sigma \]

\[ \mathcal{E}[\text{if } E_1 \text{ then } E_2 \text{ else } E_3] \rho \sigma \]
\[ = \text{ if } \mathcal{E}[E_1] \rho \sigma \neq 0 \text{ then } \mathcal{E}[E_2] \rho \sigma \text{ else } \mathcal{E}[E_3] \rho \sigma \]
Statements:

\[
\mathcal{C}[C_1; C_2] \rho \sigma = \text{let } \sigma' = \mathcal{C}[C_1] \rho \sigma \\
\text{in } \mathcal{C}[C_2] \rho \sigma' \\
\text{end}
\]

\[
\mathcal{C}[\text{if } E \text{ then } C_1 \text{ else } C_2] \rho \sigma \\
= \text{if } \mathcal{E}[E] \rho \sigma \neq 0 \text{ then } \mathcal{C}[C_1] \rho \sigma \text{ else } \mathcal{C}[C_2] \rho \sigma
\]

\[
\mathcal{C}[L := E] \rho \sigma \\
= \text{let } \alpha = \mathcal{L}[L] \rho \sigma; n = \mathcal{E}[E] \rho \sigma \\
\text{in } \sigma[\alpha \mapsto n] \\
\text{end}
\]

\[
\mathcal{C}[\text{new } I = E \text{ in } C \text{ end}] \rho \sigma \\
= \text{let } \alpha = \text{nextloc}(\sigma); n = \mathcal{E}[E] \rho \sigma \\
\text{in } \mathcal{C}[C] \ (\rho[I \mapsto \alpha]) \ (\sigma[\alpha \mapsto n]) \\
\text{end}
\]

Programs:

\[
\mathcal{P}[\text{program}(I); C \text{ end.}] n \\
= \text{let } \rho(J) = \bot; \sigma(A) = \text{unused}; \alpha = \text{nextloc}(\sigma) \\
\text{in } \text{let } \sigma_f = \mathcal{C}[C] \ (\rho[I \mapsto \alpha]) \ (\sigma[\alpha \mapsto n]) \\
\text{in } \sigma_f(\alpha) \\
\text{end}
\]

where nextloc(\sigma) = \alpha such that \sigma(\alpha) = unused.
Language combining expressions and commands

1. **Syntactic categories**

   \[ I \in \text{Ident} \quad \text{identifiers} \]
   \[ L \in \text{Lexp} \quad \text{l-values or references} \]
   \[ E \in \text{Exp} \quad \text{expressions and commands} \]
   \[ P \in \text{Prog} \quad \text{programs} \]

2. **Syntax**

   \[
   \begin{align*}
   L & \rightarrow I \\
   E & \rightarrow 0 \mid 1 \mid I \mid !L \mid E + E \mid \text{let } I = E \text{ in } E \text{ end} \\
   & \quad \mid E ; E \mid \text{if } E \text{ then } E \text{ else } E \\
   & \quad \mid L := E \mid \text{new } I = E \text{ in } E \text{ end} \\
   P & \rightarrow \text{program}(I); E \text{ end.}
   \end{align*}
   \]

3. **Value domains**

   \[
   \begin{align*}
   n & \in \text{Int} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \} \quad \text{integers} \\
   \alpha & \in \text{Loc} \quad \text{locations} \\
   \sigma & \in \text{States} = \text{Loc} \rightarrow (\text{Int} + \text{unused}) \quad \text{states} \\
   \rho & \in \text{Env} = \text{Ident} \rightarrow (\text{Int} + \text{Loc} + \bot) \quad \text{environments}
   \end{align*}
   \]
4. Semantic functions

\[ L : \text{Lexp} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{Loc} \]
\[ M : \text{Exp} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{Int} \times \text{States} \]
\[ P : \text{Prog} \rightarrow \text{Int} \rightarrow \text{Int} \]

L-values or references:

\[ L[I] \rho \sigma = \rho(I) \]

Expressions and commands:

\[ M[0] \rho \sigma = \langle 0, \sigma \rangle \]
\[ M[1] \rho \sigma = \langle 1, \sigma \rangle \]
\[ M[I] \rho \sigma = \langle \rho(I), \sigma \rangle \]
\[ M[!L] \rho \sigma = \langle \sigma(L[L] \rho \sigma), \sigma \rangle \]

\[ M[E_1 + E_2] \rho \sigma \]
\[ = \text{let } \langle n, \sigma' \rangle = M[E_1] \rho \sigma; \langle m, \sigma'' \rangle = M[E_2] \rho \sigma' \]
\[ \text{in } \langle n + m, \sigma'' \rangle \]
\[ \text{end} \]

\[ M[\text{if } E_1 \text{ then } E_2 \text{ else } E_3] \rho \sigma \]
\[ = \text{let } \langle n, \sigma' \rangle = M[E_1] \rho \sigma \]
\[ \text{in } \text{if } n \neq 0 \text{ then } M[E_2] \rho \sigma' \text{ else } M[E_3] \rho \sigma' \]
\[ \text{end} \]
\( \mathcal{M}[\text{let } I = E_1 \text{ in } E_2 \text{ end}] \rho \sigma \)

\[ = \text{let } \langle n, \sigma' \rangle = \mathcal{M}[E_1] \rho \sigma \]
\[ \text{ in } \mathcal{M}[E_2] (\rho[I \mapsto n]) \sigma' \]
\[ \text{ end} \]

\( \mathcal{M}[E_1; E_2] \rho \sigma \)

\[ = \text{let } \langle n, \sigma' \rangle = \mathcal{M}[E_1] \rho \sigma \]
\[ \text{ in } \mathcal{M}[E_2] \rho \sigma' \]
\[ \text{ end} \]

\( \mathcal{M}[L := E] \rho \sigma \)

\[ = \text{let } \alpha = \mathcal{L}[L] \rho \sigma; \langle n, \sigma' \rangle = \mathcal{M}[E_1] \rho \sigma \]
\[ \text{ in } \langle n, \sigma'[\alpha \mapsto n] \rangle \]
\[ \text{ end} \]

\( \mathcal{M}[\text{new } I = E_1 \text{ in } E_2 \text{ end}] \rho \sigma \)

\[ = \text{let } \langle n, \sigma' \rangle = \mathcal{M}[E_1] \rho \sigma; \alpha = \text{nextloc}(\sigma) \]
\[ \text{ in } \mathcal{M}[E_2] (\rho[I \mapsto \alpha]) (\rho[I \mapsto n]) \]
\[ \text{ end} \]
Language with functions

1. **Syntactic categories**

   \[ I \in \text{Ident} \quad \text{identifiers} \]
   \[ L \in \text{Lexp} \quad \text{l-values or references} \]
   \[ E \in \text{Exp} \quad \text{expressions and commands} \]
   \[ P \in \text{Prog} \quad \text{programs} \]

2. **Syntax**

   \[
   \begin{align*}
   L & \rightarrow I \\
   E & \rightarrow 0 | 1 | I | !L | E + E | \textbf{let } I = E \textbf{ in } E \textbf{ end} \\
   & \quad | E ; E | \textbf{if } E \textbf{ then } E \textbf{ else } E \\
   & \quad | L := E | \textbf{new } I = E \textbf{ in } E \textbf{ end} \\
   & \quad | \textbf{function } I(I) = E \textbf{ in } E \textbf{ end} | \textbf{call } I(E) \\
   P & \rightarrow \text{program}(I); E \textbf{ end}.
   \end{align*}
   \]

3. **Value domains**

   \[
   \begin{align*}
   n & \in \text{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \quad \text{integers} \\
   \alpha & \in \text{Loc} \quad \text{locations} \\
   \sigma & \in \text{States} = \text{Loc} \rightarrow (\text{Int} + \text{unused}) \quad \text{states} \\
   f & \in \text{Func} = \text{States} \rightarrow \text{Int} \rightarrow \text{Int} \times \text{States} \quad \text{functions} \\
   \text{Denote} & = \text{Int} + \text{Loc} + \text{Func} \quad \text{values} \\
   \rho & \in \text{Env} = \text{Ident} \rightarrow (\text{Denote} + \bot) \quad \text{environments}
   \end{align*}
   \]
4. Semantic functions

\[ L : \text{Lexp} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{Loc} \]
\[ M : \text{Exp} \rightarrow \text{Env} \rightarrow \text{States} \rightarrow \text{Int} \times \text{States} \]
\[ P : \text{Prog} \rightarrow \text{Int} \rightarrow \text{Int} \]

See semantic functions of example 8. In addition we have:

\[
M[\text{function } I_1(I_2) = E_1 \text{ in } E_2 \text{ end }] \rho \sigma
= \text{let } f' n = M[E_1] (\rho[I_2 \mapsto n]) \sigma'
  \text{ in } M[E_2] (\rho[I_1 \mapsto f]) \sigma
\text{ end}
\]

\[
M[\text{call } I(E)] \rho \sigma
= \text{let } \langle n, \sigma' \rangle = M[E] \rho \sigma; f = \rho(I)
  \text{ in } f \sigma' n
\text{ end}
\]
Chapter 9  Axiomatic Semantics

Introduction

Often one is interested in partial correctness properties of programs:

A program is partially correct, with respect to a precondition and a postcondition, if whenever the initial state fulfils the precondition and the program terminates, then the final state is guaranteed to fulfil the postcondition.

Consider for example the program

\[
z := x; \; x := y; \; y := z
\]

For this program, we have the partial correctness property

\[
\{x = n \land y = m\} \quad z := x; \; x := y; \; y := z \quad \{y = n \land x = m\}
\]

where \(x = n \land y = m\) is the precondition and \(y = n \land x = m\) is the postcondition.

The state \(\sigma = \{\langle x, 5 \rangle, \langle y, 7 \rangle, \langle z, 0 \rangle\}\) satisfies the precondition \((n = 5\) and \(m = 7)\). When we have proved the partial correctness property we can deduce that if the program terminates then it will do so in a state where \(y = 5\) and \(x = 7\).

Note: the partial correctness property does not ensure that the program will terminate.
The axiomatic semantics provides a *logical system* for proving partial correctness properties of individual programs.

The following *proof tree* expresses the proof of the partial correctness of the example program:

\[
\begin{array}{c}
\{p_0\} \quad z := x \quad \{p_1\} \\
\{p_0\} \quad z := x \quad x := y \quad \{p_2\} \\
\{p_0\} \quad z := x; x := y; y := z \quad \{p_3\}
\end{array}
\]

where

\[
\begin{align*}
p_0 &= x = n \& y = m \\
p_1 &= z = n \& y = m \\
p_2 &= z = n \& x = m \\
p_3 &= y = n \& x = m
\end{align*}
\]

The logical system is a specification of certain aspects of the semantics. Although the following example programs behave differently, they have the same partial correctness properties:

\[
\begin{align*}
\{x = n \& y = m\} & \quad z := x; x := y; y := z \quad \{y = n \& x = m\} \\
\{x = n \& y = m\} & \quad \text{if } x = y \text{ then skip else } (z := x; x := y; y := z) \\
\{y = n \& x = m\} & \quad \text{while TRUE do skip end} \quad \{y = n \& x = m\}
\end{align*}
\]
First-order predicate logic

- Terms are inductively defined by
  1. \( x \), if \( x \) is a variable
  2. \( n \), if \( n \) is a number
  3. \( -t_1 \), \( t_1 + t_2 \), \( t_1 - t_2 \), \( t_1 \cdot t_2 \) and \( t_1/t_2 \), if \( t_1 \) and \( t_2 \) are terms

- Formulas are inductively defined by
  1. TRUE and FALSE
  2. \( t_1 = t_2 \), \( t_1 \neq t_2 \), \( t_1 < t_2 \), \( t_1 \leq t_2 \), \( t_1 > t_2 \), and \( t_1 \geq t_2 \), if \( t_1 \) and \( t_2 \) are terms
  3. \( \neg \phi \), \( \phi \land \psi \), \( \phi \lor \psi \), and \( \phi \rightarrow \psi \), if \( \phi \) and \( \psi \) are formulas
  4. \( \forall x. \phi(x) \) and \( \exists x. \phi(x) \), if \( \phi \) is a formula possibly containing \( x \) as a free variable

Symbols \( \forall \) and \( \exists \) are called the universal and existential quantifiers, respectively. The variable next to a quantifier is bound to that quantifier (cf. lambda abstractions).

\[
\begin{align*}
\forall x. (x > 0 \Rightarrow -x < 0) \\
\forall x. x = x \\
\forall x. \forall y. (x \geq 0 \& \land y \geq 0 \Rightarrow x \cdot y \geq 0) \\
\exists x. x > 0 \\
\exists x. x = 3 \\
\forall x. (\exists y. x = y)
\end{align*}
\]
We will use first-order predicate logic to characterize machine states.

- The memory of a machine is a collection of labeled cells
- The contents of a labeled cell is a state

We can view a specific machine state as a function from names to values. For example, let $\sigma = \{\langle x, 3 \rangle, \langle y, 2 \rangle\}$ then $\sigma(x) = 3$, i.e. $x$ has the value 3 in state $\sigma$.

A first-order predicate logic formula can be used to characterize a set of states. For example

- $x = 3$ characterizes all those states $S \subseteq \Sigma$ in which the value of the memory cell is 3:
  $$S = \{\sigma \in \Sigma \mid \sigma(x) = 3\}$$

- $x = 3 \implies y = 2$ characterizes all states $S \subseteq \Sigma$ in which if $x = 3$ then $y = 2$:
  $$S = \{\sigma \in \Sigma \mid \sigma(x) \neq 3 \lor \sigma(y) = 2\}$$

- $\exists x. x = 3$ characterizes all states $S \subseteq \Sigma$ such that there exists some value equal to 3:
  $$S = \{\sigma \in \Sigma \mid \exists x. x = 3\} = \{\sigma \in \Sigma \mid 3 = 3\} = \Sigma$$

(Only the free variables of formulas are involved in the selection of states.)

where $\Sigma$ is the set of all possible states of a machine.
The execution of a construct $S$ in an imperative programming language can be described by the state obtained after executing $S$ given some initial state, i.e. the program construct $S$ is a state transformer.

A **Hoare triple**

$\{P\} \ S \ \{Q\}$

consists of formulas $P$ and $Q$ and a program segment $S$. Formula $P$ is the **precondition** and $Q$ is the **postcondition** of $S$.

**Partial correctness:**
We say that the triple $\{P\} \ S \ \{Q\}$ is **valid** if execution of $S$ is begun in any state satisfying precondition $P$ and ends in a state satisfying postcondition $Q$ (if $S$ terminates).

Example valid Hoare triples are

$\{2 = 2\} \ x := 2 \ \{x = 2\}$

$\{\text{TRUE}\} \ \text{if } B \ \text{then } x := 2 \ \text{else } x := 1 \ \{x = 1 \lor x = 2\}$

$\{x = 0\} \ \text{while } x = 0 \ \text{do } y := 1 \ \text{end} \ \{y = 1000000\}$

Examples of non-valid Hoare triples are

$\{2 = 2\} \ x := 2 \ \{y = 2\}$

$\{\text{TRUE}\} \ \text{if } B \ \text{then } x := 2 \ \text{else } x := 1 \ \{x = 1 \land x = 2\}$

$\{x > 0\} \ \text{while } x = 0 \ \text{do } y := 1 \ \text{end} \ \{x = 0\}$
Axiomatic semantics gives a semantic description of the program constructs using Hoare triples.

- Hoare triples are especially useful in proving partial correctness of programs, because proof systems exist for deriving Hoare triples.
- Knowing that a Hoare triple is valid guarantees that the post-condition is established, *if* the program terminates.
- Software is available that can provide a formal verification of partial correctness of a (not too large) program using Hoare triples.
- A *total correctness* semantics, where termination is assured instead of assumed, is possible but the logic involved is complex.
Syntax of the `while`-language:

\[
S \rightarrow V := T \\
\quad \text{if } B \text{ then } S \text{ else } S \\
\quad \text{while } B \text{ do } S \text{ end} \\
\quad S ; S \\
\quad \text{skip}
\]

\[
V \rightarrow \text{any of the variables in the logic specification}
\]

\[
B \rightarrow \text{a quantifier-free formula}
\]

\[
T \rightarrow \text{any term from first-order predicate logic}
\]

Some example programs

\[
z := x; \ x := y; \ y := z
\]

\[
\text{if } x = y \text{ then skip else } x := y
\]

\[
z := 0; \ n := y; \ \text{while } n > 0 \text{ do } z := z + x; \ n := n - 1 \text{ end}
\]
A Post system for deriving Hoare triples from \textit{while} programs is created by defining rules for each programming construct.

The \textit{assignment rule} is

\[
\{Q[V := T]\} \quad V := T \quad \{Q\}
\]

where $Q$ is any formula, $V$ is a program variable, and $T$ is a term.

The notation $Q[V := T]$ replaces all free occurrences of $V$ with $T$, just like the substitution algorithm used in lambda calculus. (Note: the quantifiers have bound variables, e.g. $\forall x. x > 0$ has a bound variable $x$.)

For example, let $Q$ be $x = 2$, $T$ be 2 and $V$ be $x$, then

\[
\{2 = 2\} \quad x := 2 \quad \{x = 2\}
\]

derives a valid Hoare triple from the Post system.

Let $Q$ and $V$ be as above. Let $T$ be $x + 1$. Then

\[
\{x + 1 = 2\} \quad x := x + 1 \quad \{x = 2\}
\]
The rule of assignment appears backward: One picks the postcondition and determines the precondition.

When the program segment $S$ and postcondition $Q$ are given, precondition $P$ can be determined.

A trivial solution is $P = \text{FALSE}$, because every program $S$ transforms an input state from an *empty set* to an output state characterized by $Q$.

The *weakest precondition* $\text{wp}(S, Q)$ is the precondition defining the largest set of legal input states such that the postcondition $Q$ is guaranteed to hold after executing $S$.

For the assignment rule, $\text{wp}(V := T, Q) = Q[V := T]$. For example

$$
\{x = x \& x > 0\} \quad y := x \quad \{y = x \& x > 0\}
$$

Note: $Q$ can be arbitrary and may even have no relation to the variable being assigned. For example

$$
\{x > 0\} \quad y := 7 \quad \{x > 0\}
$$

is a valid Hoare triple, because the partial correctness property merely states that if $S$ terminates given precondition $P$, then the postcondition $Q$ must hold.
When the precondition $P$ and program segment $S$ are given, postcondition $Q$ can be determined.

A trivial solution is $Q = \text{TRUE}$, because every program $S$ transforms an input state characterized by $P$ to an output state that is certainly in the set characterized by $\text{TRUE}$.

The *strongest postcondition* $\text{sp}(P, S)$ is the postcondition defining the smallest set of output states that are guaranteed to hold after executing $S$ given precondition $P$.

An assignment rule in which the postcondition is determined from program $S$ and precondition $P$ is

$$\{P\} \quad V := T \quad \{\exists y. P[V := y] \land V = T[V := y]\}$$

Because this assignment rule is more difficult to work with, the former rule (using the weakest precondition) is preferred.
Let’s say that the set of possible input states to an assignment is a subset of the states described by the precondition. Obviously, it is still legal to execute the assignment.

Consider program $y := x$ with postcondition $y = x \& y > 0$. Then, by the assignment rule

$$P = \text{wp}(y := x, y = x \& y > 0) = (y = x \& y > 0)[y := x]$$

we obtain the Hoare triple

$$\{x = x \& x > 0\} \quad y := x \quad \{y = x \& y > 0\}$$

Now, suppose that the actual input states are characterized by $P'$ being $x > 0 \& y = 2$.

Observe that the set of states characterized by $P'$ is a true subset of the states characterized by $P$ (note that for $P$ we have $x = x \& x > 0 \equiv x > 0$). So, we can derive a valid hoare triple

$$\{x > 0 \& y = 2\} \quad y := x \quad \{y = x \& y > 0\}$$

The generalization of this principle leads to the rule of precondition strengthening:

$$P' \Rightarrow P \quad \{P\} \quad S \quad \{Q\}$$

$$\{P'\} \quad S \quad \{Q\}$$

Note that logical implication $\Rightarrow$ means

*stronger condition $\Rightarrow$ weaker condition*
Similarly, we have the rule of postcondition weakening

\[
\{P\} \quad S \quad \{Q\} \quad Q \Rightarrow Q' \\
\{P\} \quad S \quad \{Q'\}
\]

We will see examples in which this rule is essential to use.

The rules of preconditioning strengthening and postcondition weakening can be combined into the rule of consequence:

\[
P' \Rightarrow P \quad \{P\} \quad S \quad \{Q\} \quad Q \Rightarrow Q' \\
\{P'\} \quad S \quad \{Q'\}
\]

A summary of relationships between conditions and states:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Logic</th>
<th>Set of States</th>
</tr>
</thead>
<tbody>
<tr>
<td>weakening</td>
<td>removing tests</td>
<td>enlarging</td>
</tr>
<tr>
<td>strengthening</td>
<td>adding tests</td>
<td>restricting</td>
</tr>
</tbody>
</table>
Ch. 9  while-Language: Skip

The *skip rule* is

\[ \{ Q \} \text{ skip } \{ Q \} \]

Which makes sense, because this no-operation does not have any side effects.

By definition, the weakest precondition of the *skip* statement is

\[ \text{wp}(\text{skip}, Q) = Q \]
The conditional rule is
\[
\begin{array}{c}
\{B & P\} & S_1 & \{Q\} & \{-B & P\} & S_2 & \{Q\} \\
\{P\} & \text{if } B \text{ then } S_1 \text{ else } S_2 & \{Q\}
\end{array}
\]

Consider for example program
\[
\text{if } x > 0 \text{ then } y := x \text{ else } y := 0
\]
Suppose we want to prove the postcondition \( Q = (y \geq 0) \).

We start with applying the conditional rule
\[
\begin{array}{c}
\{x > 0 & P\} & y := x & \{y \geq 0\} & \{-x > 0 & P\} & y := 0 & \{y \geq 0\} \\
\{P\} & \text{if } x > 0 \text{ then } y := x \text{ else } y := 0 & \{y \geq 0\}
\end{array}
\]

Now, we can try to apply the rule of assignment
\[
\begin{array}{c}
\{x \geq 0\} & y := x & \{y \geq 0\} & \{0 \geq 0\} & y := 0 & \{y \geq 0\} \\
\{x > 0 & P\} & y := x & \{y \geq 0\} & \{-x > 0 & P\} & y := 0 & \{y \geq 0\} \\
\{P\} & \text{if } x > 0 \text{ then } y := x \text{ else } y := 0 & \{y \geq 0\}
\end{array}
\]

The rule of precondition strengthening gives us the answer for the two branches:
\[
x > 0 & P \Rightarrow x \geq 0 \quad \{x \geq 0\} & y := x & \{y \geq 0\}
\]
and
\[
x \leq 0 & P \Rightarrow 0 \geq 0 \quad \{0 \geq 0\} & y := 0 & \{y \geq 0\}
\]

There are many solutions for \( P \) satisfying these two rules. The following definition gives the weakest precondition
\[
P = \wp(\text{if } B \text{ then } S_1 \text{ else } S_2, Q)
\]
\[
= \neg B \lor \wp(S_1, Q) \& (B \lor \wp(S_2, Q)) \quad \text{(Definition)}
\]
\[
= (x \leq 0 \lor x \geq 0) \& (x > 0 \lor 0 \geq 0)
\]
\[
= \text{TRUE} \& \text{TRUE} = \text{TRUE}
\]
The composition rule is
\[
\{P\} \quad S_1 \quad \{Q\} \quad S_2 \quad \{R\}
\]
\[
\{P\} \quad S_1; \quad S_2 \quad \{R\}
\]

The weakest precondition \( P \) is defined by
\[
P = \text{wp}(S_1; \ S_2, R) = \text{wp}(S_1, Q) = \text{wp}(S_1, \text{wp}(S_2, R))
\]

Consider for example program
\[
z := x; \ x := y
\]
Suppose we are given the postcondition \( R = (z > 0 \& x = y) \).

We have
\[
\{P\} \quad z := x + 1 \quad \{Q\} \quad \{Q\} \quad x := y \quad \{z > 0 \& x = y\}
\]
\[
\{P\} \quad z := x + 1; \ x := y \quad \{z > 0 \& x = y\}
\]
Using the assignment rule for \( S_2 \), we obtain
\[
Q = \text{wp}(x := y, z > 0 \& x = y) = (z > 0 \& y = y) = (z > 0)
\]
And for the composition we find
\[
P = \text{wp}(z := x + 1, Q) = (x + 1 > 0)
\]
In other words, in the proof tree we have the axiom
\[
\{x + 1 > 0\} \quad z := x + 1 \quad \{z > 0\}
\]
for \( S_1 \) and the axiom
\[
\{z > 0\} \quad x := y \quad \{z > 0 \& x = y\}
\]
for \( S_2 \).
The *simultaneous* assignment statement assigns values to variables in “parallel”. For example,

\[(x, y) := (1, 2)\]

assigns 1 to \(x\) and 2 to \(y\) in one statement.

The assignment is performed by copying the values on the right-hand side and assigning the copies simultaneously to the variables on the left-hand side.

For example, we can swap the values of \(x\) and \(y\) by

\[(x, y) := (y, x)\]

The application of the assignment rule on \((x, y) := (y, x)\), for example, given postcondition \(Q = (x > 0 \& y = 1)\) gives

\[P = \text{wp}((x, y) := (y, x), Q)\]

\[= Q[(x, y) := (y, x)]\]

\[= (x > 0 \& y = 1)[(x, y) := (y, x)]\]

\[= (y > 0 \& x = 1)\]

Note that the substitution of \(x\) and \(y\) in conditions proceeds *independently in parallel*.

The following substitution, for example, is illegal:

\[(x > 0 \& y = 1)[(x, y) := (y, x)]\]

\[= ((x > 0 \& y = 1)[x := y])[y := x]\]

\[= (y > 0 \& y = 1)[y := x]\]

\[= (x > 0 \& x = 1)\]
Consider program $S$

$$S_1: \text{if } y \leq x \& y \leq z \text{ then } (x, y) := (y, x)$$
$$\quad \text{else if } z \leq x \& z \leq y \text{ then } (x, z) := (z, x) \text{ else skip}$$
$$S_2: \text{if } y \geq z \text{ then } (y, z) := (z, y) \text{ else skip}$$

where $S_1$ and $S_2$ label the composition of $S$ by $S_1; S_2$.

We want to prove that

$$\{\text{TRUE}\} \ S \ \{x \leq y \& y \leq z\}$$

Again, we proceed backwards:

$$\wp(S_2, x \leq y \& y \leq z)$$
$$= (\neg y \geq z \lor \wp((y, z) := (z, y), x \leq y \& y \leq z))$$
$$\quad \& (y \geq z \lor \wp(\text{skip}, x \leq y \& y \leq z))$$
$$= (y \leq z \lor x \leq z \& z \leq y \& y \leq z) \lor (y \geq z \lor x \leq y \& y \leq z)$$
$$= (y \leq z \& y \geq z)$$
$$\lor (y \leq z \& x \leq y \& y \leq z)$$
$$\lor (x \leq z \& z \leq y \& y \geq z)$$
$$\lor (x \leq z \& z \leq y \& x \leq y \& y \leq z)$$
$$= (\text{FALSE})$$
$$\lor (y \leq z \& x \leq y)$$
$$\lor (x \leq z \& z \leq y)$$
$$\lor (x \leq z \& y = z)$$
$$= x \leq y \& (y \leq z \lor x \leq z \& z \leq y)$$
$$= x \leq y \& x \leq z$$
Now for $S_1$ we find:

\[
wp(S_1, x \leq y \& x \leq z) \\
= (\neg (y \leq x \& y \leq z) \lor wp((x,y) := (y,x), x \leq y \& x \leq z)) \\
\land (y \leq x \& y \leq z) \\
\lor wp(if z \leq x \& z \leq y \\
then (x,z) := (z,x) \\
else \text{skip} \\
, x \leq y \& x \leq z)) \\
= (y > x \lor y > z \lor y \leq x \& y \leq z) \\
\land (y \leq x \& y \leq z) \\
\lor (\neg (z \leq x \& z \leq y) \lor wp((x,z) := (z,x), x \leq y \& x \leq z)) \\
\land (z \leq x \& z \leq y \lor wp(\text{skip}, x \leq y \& x \leq z)) \\
= (x < y \lor z < y \lor y \leq x \& y \leq z) \\
\land (y \leq x \& y \leq z) \\
\lor (z > x \lor z > y \lor z \leq y \& z \leq x) \\
\land (z \leq x \& z \leq y \lor x \leq y \& x \leq z)) \\
= (x < y \lor z < y \lor y \leq x \& y \leq z) \\
\land (y \leq x \& y \leq z) \\
\lor (x < z \lor y < z \lor z \leq y \& z \leq x) \\
\land (z \leq x \& z \leq y \lor x \leq y \& x \leq z)) \\
= x \leq y \& x \leq z \lor y \leq x \& y \leq z \lor z \leq x \& z \leq y \\
= \text{TRUE}
\]
In order to define the rule for the while loop in our example language, we first need to have a look at loop invariants.

A loop invariant condition is a logical expression that is true prior the loop, during the loop, and after the loop.

Here is an “every day” example: shopping.

We have a list of groceries we want to purchase.

cart := empty;
{groceries wanted = groceries unchecked + groceries in cart}
while grocery list not empty do
    {groceries wanted = groceries unchecked + groceries in cart and not empty list}
    add grocery to cart;
    check grocery on list
    {groceries wanted = groceries unchecked + groceries in cart}
end;
{groceries wanted = groceries unchecked + groceries in cart and empty list}

Invariant in this loop is

{groceries wanted = groceries unchecked + groceries in cart}

because each time we put a grocery in the cart we check it on the list.
The **while rule** is

\[
\begin{array}{c}
\{B \& I\} \quad S \quad \{I\} \\
\{I\} \quad \text{while } B \text{ do } S \text{ end} \quad \{\neg B \& I\}
\end{array}
\]

where \(I\) is the loop invariant.

The weakest precondition is the invariant

\[
\text{wp}(\text{while } B \text{ do } S \text{ end}, Q) = I
\]

where the invariant must fulfil \(B \& I = \text{wp}(S, I)\).

For example,

\[
z := 0; \ n := y; \ \text{while } n > 0 \text{ do } z := z + x; \ n := n - 1 \ \text{end}
\]

has invariant

\[
z = x \ast (y - n) \& n \geq 0
\]

The program with the invariant is

\[
\{y \geq 0\}
\]

\[
z := 0; \ n := y;
\]

\[
\text{while } \ n > 0 \text{ do }
\]

\[
\{z = x \ast (y - n) \& n > 0\}
\]

\[
z := z + x; \ n := n - 1
\]

\[
\{z = x \ast (y - n) \& n \geq 0\}
\]

\[
\text{end}
\]

\[
\{z = x \ast y\}
\]

where \(y \geq 0\) is a precondition necessary to guarantee that the loop terminates.
We will examine the axiomatic semantics for procedures developed by Alain Martin.

We extend the syntax of the while-language with

\[
\begin{align*}
S & \rightarrow \text{proc } I ( \text{args} ) ; S \\
S & \rightarrow \text{call } I ( \text{terms} ) \\
\text{args} & \rightarrow \text{arg} , \text{args} \\
\text{arg} & \rightarrow \text{in } I \\
& \quad \text{inout } I \\
& \quad \text{out } I \\
\text{terms} & \rightarrow T , \text{terms} \\
& \quad T
\end{align*}
\]

where \( S \) is a statement of the while language, \( T \) is a term (expression), and \( I \) is an identifier.

Parameter passing is either \textit{call-by-value}, \textit{copy-in/copy-out}, or \textit{copy-out}:

- \textbf{in} \( a \)
  - means argument \( a \) is passed as call-by-value

- \textbf{inout} \( a \)
  - means argument \( a \) is passed as copy-in/copy-out

- \textbf{out} \( a \)
  - means argument \( a \) is passed as copy-out
Here are some simple example procedure definitions. The first increments an integer, the second computes faculty numbers, the third performs a swap, and the fourth sorts three integers in ascending order:

```plaintext
proc inc(in x, out z);
  z := x+1

proc fac(in x, out f);
  n := x;
  f := 1;
  while n>0 do
    f := n*f;
    n := n-1
  end

proc swap(inout x, inout y);
  (x,y) := (y,x)

proc sort(inout x, inout y, inout z);
  if y=x & y=z then
    call swap(x, y)
  else if z=x & z=y then
    call swap(x, z)
  else
    skip;
  if y=z then
    call swap(y, z)
  else
    skip
```
Assume we have a procedure \( p \) defined as

\[
\text{proc } p(\text{in } x, \text{inout } y, \text{out } z); S
\]

Then, the proof rule for the call

\[
\text{call } p(a, b, c)
\]

is

\[
Q \land A \Rightarrow R[(b, c) := (y, z)] \quad \{P\} \quad S \quad \{Q\}
\]

\[
\{P[(x, y) := (a, b)] \land A[x := a]\} \quad \text{call } p(a, b, c) \quad \{R\}
\]

with the following restrictions:

- the formal \text{in} argument \( x \) is not changed in body \( S \) (\( x \) is passed call-by-value)
- the postcondition \( Q \) has only variables \( x, y, \) and \( z \) free (the postcondition cannot state properties of global variables)
- the actual \text{inout} argument \( b \) and actual \text{out} argument \( c \) are variables
- the predicate \( A \) does not have the variables \( y \) and \( z \) free

Note that different procedures require different proof rules depending on the \text{in}, \text{inout}, and \text{out} combinations of arguments. The rule above is an instance of a general case with all three types of arguments.
Consider the procedure

\[\text{proc inc(in } x, \text{ out } z);\]
\{TRUE\} \quad z := x + 1 \quad \{z = x + 1\}\]

The Hoare triple \{TRUE\} \quad z := x + 1 \quad \{z = x + 1\} is valid.

We want to prove that

\{a = a_0\} \quad \text{call inc}(a, a) \quad \{a = a_0 + 1\}\]

is a valid triple.

Since \(x\) is an in argument and \(z\) is an out argument, we have the rule instance
\[z = x + 1 \& A \Rightarrow z = a_0 + 1 \quad \{\text{TRUE}\} \quad z := x + 1 \quad \{z = x + 1\}\]

\{\text{TRUE} \& A[x := a]\} \quad \text{call inc}(a, a) \quad \{a = a_0 + 1\}\]

Now we have to find a suitable \(A\) that satisfies the implication in the rule.

If we take \(A\) to be \(x = a_0\), then the premis of the rule is a tautology and we conclude that

\{a = a_0\} \quad \text{call inc}(a, a) \quad \{a = a_0 + 1\}\]

is a valid Hoare triple.
Ch. 6  Introduction to Prolog

The objective of this quick introduction to Prolog is twofold:

- to learn the principles of declarative programming
- and to demonstrate denotational semantics written in Prolog for the while language (of Section 8.10).

Sofar, we have seen examples of functional programming in the form of lambda calculus and examples of imperative programming with exercises in denotational and axiomatic semantics.

Prolog is concerned with declarative programming in which the solution is specified, rather than the process of deriving the solution.

Prolog is ideal for prototyping in which efficiency is not an issue.
Programming in Prolog involves

- **objects** which are terms composed of atoms, variables, functors, and lists that make up data structures
- **relationships** between objects

The following syntactic conventions for objects (terms) are used. A term is

- **an atom**: a number or a name starting with a lower case letter, e.g. `cop5025`
- **a variable**: a name starting with an upper case letter, e.g. `X`
- **a list**: either empty `[]`, or a list of terms $t_i$ written as $[t_1, t_2, \ldots, t_n]$
- **a functor**: this is an unevaluated function symbol $f$ applied to terms $t_i$ and written $f(t_1, t_2, \ldots, t_n)$. (The functor name must start with a lower case letter.)

Examples of terms:

```
john
person(john, smith)
book(jane_eyre, charlotte_bronte)
[gold, person(john, smith), [1, 2], money]
```
Relationships between terms are defined by facts and rules: programming in Prolog consists of

- declaring \textit{facts} about objects and relationships
- defining \textit{rules} about objects and relationships
- and \textit{queries} asking questions about objects and relationships

Examples of declaring facts about objects:

\begin{verbatim}
valuable(gold).
valuable(money).
valuable(book(jane_eyre, charlotte_bronte)).
father(john, mary).
king(john, france).
gives(john, paper, mark).
gives(john, gold, mark).
gives(john, book(jane_eyre, charlotte_bronte), mark).
\end{verbatim}

These facts are stored in Prolog’s database.

Now, we can query the system with some questions:

\begin{verbatim}
?- father(john, mary).
Yes
?- french(john).
No
\end{verbatim}

We can use variables like \textit{X} for unknowns:

\begin{verbatim}
?- valuable(X).
X = gold
?- gives(X, paper, Y).
X = john
Y = mark
\end{verbatim}
The previous query examples are simple to answer. Instead, we can ask the system to fulfill the conjunction of two or more goals simultaneously. For example:

\[-\text{gives(john, X, mark), valuable(X).}\]
\[X = \text{gold}\]

The first and second goal must both be valid for the same unknown value \(X\). The system gives a solution \(X = \text{gold}\).

The variable \(X\) is \textit{instantiated} to \texttt{gold} and cannot change while the goals are processed.

This is different from querying two separate and therefore unrelated goals:

\[-\text{gives(john, X, mark).}\]
\[X = \text{paper}\]
\[?- \text{valuable(X).}\]
\[X = \text{gold}\]

More examples:

\[-\text{gives(X, paper, Y), king(X, france).}\]
\[X = \text{john}\]
\[Y = \text{mark}\]
\[?- \text{father(john, X), valuable(X).}\]
\[\text{No}\]
Interestingly in Prolog, we can use variables for placeholders of values within terms. For example, we can ask:

```prolog
?- gives(john, book(X, Y), mark).
X = jane_eyre
Y = charlotte_bronte
```

That is, we are looking for a book that *john* gave *mark* with some unknown title *X* and author *Y*.

Prolog uses term *unification* in matching terms from facts and queries. Unification attempts to instantiate variables in matching two terms.

<table>
<thead>
<tr>
<th>term</th>
<th>term</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>john</td>
<td>john</td>
<td>success</td>
</tr>
<tr>
<td>john</td>
<td>mary</td>
<td>failure</td>
</tr>
<tr>
<td>john</td>
<td>X</td>
<td>X=john</td>
</tr>
<tr>
<td>X</td>
<td>mark</td>
<td>X=mark</td>
</tr>
<tr>
<td>book(X, Y)</td>
<td>book(jane_eyre, charlotte_bronte)</td>
<td>X=jane_eyre</td>
</tr>
<tr>
<td>person(X, smith)</td>
<td>person(john, smith)</td>
<td>X=john</td>
</tr>
<tr>
<td>person(X, X)</td>
<td>person(john, smith)</td>
<td>failure</td>
</tr>
<tr>
<td>person(X, X)</td>
<td>person(john, john)</td>
<td>X=john</td>
</tr>
<tr>
<td>[X,loves,noles]</td>
<td>[he,loves,Y]</td>
<td>X=he, Y=noles</td>
</tr>
<tr>
<td>[X,loves,noles]</td>
<td>[X,loves,X]</td>
<td>X=noles, Y=noles</td>
</tr>
</tbody>
</table>

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
Rules of a Prolog program are like definitions of queries. We can ask the system

```
?- gives(X, Y, Z), valuable(Y).
X = john
Y = gold
Z = mark
```

If this query is used a lot, we might as well add a rule to the Prolog system

```
gift(X, Y, Z) :- gives(X, Y, Z), valuable(Y).
```

and ask

```
?- gift(john, Y, mark).
Y = gold
```

In general, a Prolog rule (clause) has the form

```
⟨predicate⟩ :- ⟨comma-separated goals⟩ .
```

and should be read: “I can conclude *predicate* if all goals succeed”.

Compare this with a Post system. Actually, it is straightforward to implement a Post system in Prolog. Axioms become facts, and productions become rules. For example:

```
natnum(0). % 0 is a natural number
natnum(s(X)) :- natnum(X). % s(X) is a natural number if X is

?- natnum(s(s(s(s(0))))).
Yes.
```
Consider the following Prolog program:

(1) male(albert).
(2) male(edward).
(3) female(alice).
(4) female(victoria).
(5) parents(edward, victoria, albert).
(6) parents(alice, victoria, albert).
(7) sister(X,Y) :- female(X), parents(X,M,F), parents(Y,M,F).

When we ask

?- sister(alice, X).

The system applies *backward chaining* to find the answer:

1.  *sister(alice, X₁)* matches rule 7, $X = alice$, $Y = X₁$

2. New goals:  *female(alice), parents(alice, M, F), parents(X₁, M, F)*

3.  *female(alice)* matches rule 3

4. *parents(alice, M, F)* matches rule 6, $M = victoria$, $F = albert$

5. *parents(X₁, victoria, albert)* matches rule 5, $X₁ = edward$

At this point all goals are checked and the system deduces that $X₁ = edward$.

The search order is deterministic. Goals are checked from left to right and rules are matched from the first to the last.
From the example on the previous slide it is not difficult to see that there is another solution to the query

\[
? - \text{sister(a}l\text{ice, } X) . \\
X = \text{edward}
\]

The other solution is \(X = \text{alice}\). We can let Prolog backtrack for more solutions (by typing ;). This works as follows: if the search for rules by the system yields a matching rule, the system is able to continue from that point later on.

Consider the example murder mystery

\begin{verbatim}
murderer(X) :- hair(X, brown).
attire(mr_holman, ring).
attire(mr_pope, watch).
attire(mr_woodley, pincenez) :-
   attire(sir_raymond, tattered_cuffs).
attire(sir_raymond, pincenez) :-
   attire(mr_woodley, tattered_cuffs).
attire(X, tattered_cuffs) :- room(X, 16).
hair(X, black) :- room(X, 14).
hair(X, grey) :- room(X, 12).
hair(X, brown) :- attire(X, pincenez).
hair(X, red) :- attire(X, tattered_cuffs).
room(mr_holman, 12).
room(sir_raymond, 10).
room(mr_woodley, 16).
room(X, 14) :- attire(X, watch).
\end{verbatim}
Ch. 6  *Backtracking*

Backtracking is part of the search strategy in Prolog. The goal

```
?- murderer(X).
```

has the following execution trace, where the nesting depth is indicated by indentation:

```
murderer(X)
  hair(X, brown)
  attire(X, pincenez)
    X = mr_woodley
    attire(sir_raymond, tattered_cuffs)
      room(sir_raymond, 16)
      FAIL (no facts or rules)
      FAIL (no alternative rules)
      REDO (found one alternative rule)
    attire(X, pincenez)
      X = sir_raymond
      attire(mr_woodley, tattered_cuffs)
        room(mr_woodley, 16)
        SUCCESS
        SUCCESS: X = sir_raymond
        SUCCESS: X = sir_raymond
        SUCCESS: X = sir_raymond
        SUCCESS: X = sir_raymond
```

The goal `room(sir_raymond, 16)` fails, and the system backtracks to a point where it can try alternative rules. The **REDO** indicates where the search continues.
Sometimes, it is useful to cut the backtracking to omit the search for alternatives. This can increase efficiency, or can be used to change the semantics of the program.

An example of a cut:

```prolog
beautiful(X) :- purple(X), !, fail.
beautiful(X).
```

This defines that everything is beautiful unless it is purple.

Consider the facts:

```prolog
purple(corvette).
red(porsche).
```

A query:

```
betterful(porsche)
    purple(porsche), !, fail
    purple(porsche)
    FAIL
REDO
    beautiful(porsche)
    SUCCESS
```

Another query:

```prolog
beautiful(corvette)
    purple(corvette), !, fail
    purple(corvette)
    SUCCESS
    !
    fail
    FAIL
FAIL (backtracking is cut here)```
The examples below use the convention that \([\text{Head}|\text{Tail}]\) denotes a list with head \textbf{Head} and tail \textbf{Tail}, where tail is a list of the remaining elements. For example, \([a,b,c]\) unifies with \([X|Xs]\) giving \(X=a\) and \(Xs=[b,c]\). Note that \([X|Xs]\) unifies with singleton list \([a]\) giving \(X=a\) and \(Xs=[]\). More general, \([X,Y,...|Xs]\) denotes a list with elements \(X, Y, \ldots\), and tail \(Xs\).

Finding an element in a list:

\[
\text{member}(X, [X|Xs]).
\]
\[
\text{member}(X, [Y|Xs]) :- \text{member}(X, Xs).
\]

Appending two lists:

\[
\text{append}([], Xs, Xs).
\]
\[
\text{append}([X|Xs], Ys, [X|Zs]) :- \text{append}(Xs, Ys, Zs).
\]

Length of a list:

\[
\text{length}([], 0).
\]
\[
\text{length}([X|Xs], N) :-
    \text{length}(Xs, K),
    N is K + 1.
\]

Bubble sort:

\[
\text{bsort}(Xs, Zs) :-
    \text{append}(As, [A,B|Bs], Xs),
    A > B, !,
    \text{append}(As, [B,A|Bs], Ys),
    \text{bsort}(Ys, Zs).
\]
\[
\text{bsort}(Xs, Xs).
\]
One of the aims of the design of Prolog was parsing. The following grammar

\[
s \rightarrow np \ vp
\]
\[
np \rightarrow det \ n
\]
\[
vp \rightarrow iv
\]
\[
| tv \ np
\]
\[
det \rightarrow \text{the} | \text{a} | \text{an}
\]
\[
n \rightarrow \text{giraffe} | \text{apple}
\]
\[
iv \rightarrow \text{dreams}
\]
\[
tv \rightarrow \text{dreams} | \text{eats}
\]

can be rewritten as a Prolog *Definite Clause Grammar* (DCG)

\[
s --> np, \ vp.
\]
\[
np --> det, n.
\]
\[
vp --> iv.
\]
\[
vp --> tv, np.
\]
\[
det --> \text{[the]}.
\]
\[
det --> \text{[a]}.
\]
\[
det --> \text{[an]}.
\]
\[
n --> \text{[giraffe]}.
\]
\[
n --> \text{[apple]}.
\]
\[
iv --> \text{[dreams]}.
\]
\[
tv --> \text{[dreams]}.
\]
\[
tv --> \text{[eats]}.
\]

The DCG rules are directly compiled by the system into Prolog clauses. For example

\[
s(X, Z) :- np(X, Y), \ vp(Y, Z).
\]

The obtained Prolog program is a recursive descent parser.
We can ask the system whether or not “the giraffe dreams” is a sentence:

?- s([the, giraffe, dreams], []).  
Yes

Because Prolog is relational, for certain DCGs we can let the system generate sentences:

?- s(X, []). 
X = [the, giraffe, dreams] ;
X = [the, giraffe, dreams, the, giraffe] ;
X = [the, giraffe, dreams, the, apple] ;
X = [the, giraffe, dreams, a, giraffe] ;
X = [the, giraffe, dreams, a, apple] ;
X = [the, giraffe, dreams, an, giraffe] ;
X = [the, giraffe, dreams, an, apple] ;
X = [the, giraffe, eats, the, giraffe] ;
X = [the, giraffe, eats, the, apple] ;
X = [the, giraffe, eats, a, giraffe] ;
X = [the, giraffe, eats, a, apple] ;
X = [the, giraffe, eats, an, giraffe] ;
X = [the, giraffe, eats, an, apple] ;
X = [the, apple, dreams] ;
X = [the, apple, dreams, the, giraffe] ;
X = [the, apple, dreams, the, apple] ;
X = [the, apple, dreams, a, giraffe] ;
X = [the, apple, dreams, a, apple] ;
X = [the, apple, dreams, an, giraffe] ;
X = [the, apple, dreams, an, apple] ;
...

where typing ; after the answer reveals the alternative solutions.
We can augment the grammar with parameters to produce parse trees:

\[
\begin{align*}
  &s(s(NP, VP)) \rightarrow np(NP), vp(VP). \\
  &np(np(DET, N)) \rightarrow det(DET), n(N). \\
  &vp(vp(IV)) \rightarrow iv(IV). \\
  &vp(vp(TV, NP)) \rightarrow tv(TV), np(NP). \\
  &det(det(the)) \rightarrow [the]. \\
  &det(det(a)) \rightarrow [a]. \\
  &det(det(an)) \rightarrow [an]. \\
  &n(n(giraffe)) \rightarrow [giraffe]. \\
  &n(n(apple)) \rightarrow [apple]. \\
  &iv(iv(dreams)) \rightarrow [dreams]. \\
  &tv(tv(dreams)) \rightarrow [dreams]. \\
  &tv(tv(eats)) \rightarrow [eats]. \\
\end{align*}
\]

Parsing produces a term that represents a parse tree:

\[
?- s(Tree, [the, giraffe, dreams], []). \\
Tree = s(np(det(the), n(giraffe)), vp(iv(dreams)))
\]
Here is a DCG that produces abstract syntax trees for the while language of Section 8.10:

\[
\begin{align*}
\text{s(program(I, E)) } & \rightarrow \text{ ws, "program", ws, "("}, \text{ ws, id(I), ws, ")", ws, ";", ws, e(E), ws, "end.", ws.} \\
\text{e(E1 ; E2) } & \rightarrow \text{ e1(E1), ws, ";", ws, e(E2).} \\
\text{e(E) } & \rightarrow \text{ e1(E).} \\
\text{e1(assign(L, E)) } & \rightarrow \text{ id(L), ws, ":=", ws, e1(E).} \\
\text{e1(E) } & \rightarrow \text{ e2(E).} \\
\text{e2(E1 + E2) } & \rightarrow \text{ e3(E1), ws, "+", ws, e2(E2).} \\
\text{e2(E) } & \rightarrow \text{ e3(E).} \\
\text{e3(num(N)) } & \rightarrow \text{ num(N).} \\
\text{e3(id(I)) } & \rightarrow \text{ id(I).} \\
\text{e3(!I)) } & \rightarrow \text{ ";", ws, id(I).} \\
\text{e3(let(I, E1, E2)) } & \rightarrow \text{ "let", ws, id(I), ws, ";", ws, e(E1), ws, ";", ws, e(E2), ws, ";", ws.} \\
\text{e3(if(E1, E2, E3)) } & \rightarrow \text{ "if", ws, e(E1), ws, "then", ws, e(E2), ws, "else", ws, e(E3), ws, ";", ws.} \\
\text{e3(new(I, E1, E2)) } & \rightarrow \text{ "new", ws, id(I), ws, ";", ws, e(E1), ws, ";", ws, e(E2), ws, ";", ws.} \\
\text{e3(while(E1, E2)) } & \rightarrow \text{ "while", ws, e(E1), ws, "do", ws, e(E2), ws, "od".} \\
\text{e3(E) } & \rightarrow \text{ "(", ws, e(E), ws, ")".} 
\end{align*}
\]

This DCG parses character lists. The nonterminal \textit{ws} skips white space, \textit{id} and \textit{num} represent identifiers and integers.
Ch. 6  Denotational Semantics

\[
p(\text{program}(I, E), N, K) :- \\
\quad R = [], \\
\quad S = [], \\
\quad \text{nextloc}(S, A), \\
\quad \text{e}(E, [(I,A)|R], [(A,N)|S], X), \\
\quad X = (_,Sf), \\
\quad \text{member}((A,K), Sf).
\]

\[
e(\text{num}(N), _, S, (N,S)).
\]

\[
e(\text{id}(I), R, S, (N,S)) :- \\
\quad \text{member}((I,N), R).
\]

\[
e(\text{!(I)}, R, S, (N,S)) :- \\
\quad \text{member}((I,A), R), \\
\quad \text{member}((A,N), S).
\]

\[
e(\text{assign}(I, E), R, S, (N,T)) :- \\
\quad \text{l}(I, R, S, A), \\
\quad \text{e}(E, R, S, X), \\
\quad X = (N,S1), \\
\quad T = [(A,N)|S1].
\]

\[
e(E_1 + E_2, R, S, (N,T)) :- \\
\quad \text{e}(E_1, R, S, X), \\
\quad X = (N1,S1), \\
\quad \text{e}(E_2, R, S1, Y), \\
\quad Y = (N2,T), \\
\quad N \text{ is } N1 + N2.
\]

\[
e(\text{let}(I, E_1, E_2), R, S, (N,T)) :- \\
\quad \text{e}(E_1, R, S, X), \\
\quad X = (N1,S1), \\
\quad \text{e}(E_2, [(I,N1)|R], S1, Y), \\
\quad Y = (N,T).
\]

\[
e(E_1 ; E_2, R, S, (N,T)) :- \\
\quad \text{e}(E_1, R, S, X), \\
\quad X = (_,S1), \\
\quad \text{e}(E_2, R, S1, Y), \\
\quad Y = (N,T).
\]

\[
e(\text{if}(E_1, E_2, E_3), R, S, (N,T)) :- \\
\quad \text{e}(E_1, R, S, X), \\
\quad X = (N1,S1), \\
\quad \text{cond}(N1, E_2, E_3, R, S1, Y), \\
\quad Y = (N,T).
\]
\[ e(\text{new}(I, E_1, E_2), R, S, (N,T)) :- \]
\[ e(E_1, R, S, X), \]
\[ X = (N_1,S_1), \]
\[ \text{nextloc}(S_1, A), \]
\[ e(E_2, [(I,A)|R], [(A,N_1)|S_1], Y), \]
\[ Y = (N,T). \]
\[ e(\text{while}(E_1, E_2), R, S, (N,T)) :- \]
\[ e(E_1, R, S, X), \]
\[ X = (N_1,S_1), \]
\[ \text{while}(N_1, E_1, E_2, R, S_1, Y), \]
\[ Y = (N,T). \]
\[ l(I, R, _, A) :- \text{member}((I,A), R). \]
\[ \text{cond}(0, _, E, R, S, Y) :- !, e(E, R, S, Y). \]
\[ \text{cond}(\_, E, \_, R, S, Y) :- e(E, R, S, Y). \]
\[ \text{while}(0, \_, \_, \_, S, Y) :- !, Y = (0,S). \]
\[ \text{while}(\_, E_1, E_2, R, S, Y) :- \]
\[ e(E_2, R, S, Z), \]
\[ Z = (\_,S_1), \]
\[ e(\text{while}(E_1, E_2), R, S_1, Y). \]
\[ \text{nextloc}(S, A) :- \text{length}(S, A). \]
We can also produce machine code for a stack machine.

- push\((constant)\) push integer constant
- pop discard top of stack
- add pop top two elements and push sum
- load\((address)\) push value stored at address
- store\((address)\) store top of stack at address
- jmp\((label)\) jump to label
- jeq\((label)\) pop and if value is 0, jump
- lab\((label)\) label next instruction

Here is the compiler:

```prolog
cp(program(I, E), C) :-
    R = [],
    ce(E, [[I,0]|R], C).

ce(num(N), _, push(N)).

ce(id(I), R, push(N)) :-
    member((I,N), R).

ce(!I, R, load(A)) :-
    member((I,A), R).

ce(assign(I, E), R, C; store(A)) :-
    member((I,A), R),
    ce(E, R, C).

ce(E1 + E2, R, C1; add) :-
    ce(E1, R, C1),
    ce(E2, R, C2).

ce(E1 ; E2, R, C1; pop; C2) :-
    ce(E1, R, C1),
    ce(E2, R, C2).

ce(if(E1, E2, E3), R, C1; jeq(L1); C2; jmp(L2); lab(L1); C3; lab(L2)) :-
    ce(E1, R, C1),
    ce(E2, R, C2),
    ce(E3, R, C3).

ce(new(I, E1, E2), R, C1; store(A); pop; C2) :-
    ce(E1, R, C1),
    length(R, A),
    ce(E2, [(I,A)|R], C2).

ce(while(E1, E2), R, lab(L1); C1; jeq(L2); C2; pop; jmp(L1); lab(L2); push(0)) :-
    ce(E1, R, C1),
    ce(E2, R, C2).
```
program (n);
  new a := 0 in
  new b := 1 in
  while !n do
    new t := !a in
    a := !b; b := !t + !b; n := !n + -1
  end
  od
end;
end.

push(0);
store(1); % a := 0
pop;
push(1);
store(2); % b := 1
pop;
lab(_G1329);
load(0); % load !n
jeq(_G1337); % while !n<>0 do
load(1);
store(3); % t := !a
pop;
load(2);
store(1); % a := !b
pop;
load(3);
load(2);
add;
store(2); % b := !t + !b
pop;
load(0);
push(-1);
add;
store(0); % n := !n + -1
pop;
jmp(_G1329);
lab(_G1337); % od
push(0);
pop;
load(1);
store(0) % n := !a
To run Prolog, execute command `pl` on `xi`. You will see something similar to

```
Welcome to SWI-Prolog (Version 3.2.7)
Copyright (c) 1993-1998 University of Amsterdam. All rights reserved.

For help, use `?- help(Topic).` or `?- apropos(Word).`
```

`?-`


Try

```
sister(alice,X).
```

Note that queries must end with a period (.)

Try

```
trace,murderer(X).
```

and type `ENTER` to trace through the program. (`creep` is not meant to be insulting, except maybe for the murderer.)

Try

```
s(X,[the,giraffe,dreams],[]).
```