Formal Semantics
Formal Semantics

- At the beginning of the book we saw formal definitions of syntax with BNF
- And how to make a BNF that generates correct parse trees: “where syntax meets semantics”
- We saw how parse trees can be simplified into abstract syntax trees (AST’s)
- Now… the rest of the story: formal definitions of programming language semantics
Outline

- Natural semantics and Prolog interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions
Defining Language One

■ A little language of integer expressions:
  - Constants
  - The binary infix operators + and *, with the usual precedence and associativity
  - Parentheses for grouping

■ Lexical structure: tokens are +, *, (, ), and integer constants consisting of one or more decimal digits
Syntax: Phrase Structure

\[
\begin{align*}
<exp> & : = <exp> + <mulexp> \mid <mulexp> \\
<mulexp> & : = <mulexp> * <rootexp> \mid <rootexp> \\
<rootexp> & : = ( <exp> ) \mid <constant>
\end{align*}
\]

- (A subset of ML expressions, Java expressions, and Prolog terms)
- This grammar is unambiguous
- Both operators are left associative, and * has higher precedence than +
Parse Trees And AST’s

- The grammar generates parse trees
- The AST is a simplified form: same order as the parse tree, but no non-terminals

```
<exp>          + <mulexp>
<exp>          /<mulexp>
<mulexp>       * <rootexp>
<rootexp>      1
<rootexp>      2
               3

+    *
1    2    3
```
Continuing The Definition

- That is as far as we got in Chapters 2 and 3
- One way to define the semantics of the language is to give an interpreter for it
- We will write one in Prolog, using AST’s as input:

```
plus(const(1),
    times(const(2),
        const(3)))
```
Abstract Syntax

- Note: the set of legal AST’s can be defined by a grammar, giving the abstract syntax of the language

\[ <exp> ::= \text{plus}(<exp>,<exp>) \]
\[ | \text{times}(<exp>,<exp>) \]
\[ | \text{const}(<constant>) \]

- An abstract syntax can be ambiguous, since the order is already fixed by parsing with the original grammar for concrete syntax
Language One: Prolog Interpreter

\[
\text{val1}(\text{plus}(X,Y), \text{Value}) :- \\
\quad \text{val1}(X, \text{XValue}), \\
\quad \text{val1}(Y, \text{YValue}), \\
\quad \text{Value is XValue + YValue}. \\
\text{val1}(\text{times}(X,Y), \text{Value}) :- \\
\quad \text{val1}(X, \text{XValue}), \\
\quad \text{val1}(Y, \text{YValue}), \\
\quad \text{Value is XValue * YValue}. \\
\text{val1}(\text{const}(X), X).
\]
?- val1(const(1),X).
  X = 1.

?- val1(plus(const(1),const(2)),X).
  X = 3.

?- val1(plus(const(1),times(const(2),const(3))),X).
  X = 7.
Problems

- What is the value of a constant?
  - Interpreter says \texttt{val1(const(X),X)}.
  - This means that the value of a constant in Language One is whatever the value of that same constant is in Prolog.
  - Unfortunately, different implementations of Prolog handle this differently.
Value Of A Constant

Some Prologs treat values greater than $2^{31} - 1$ as floating-point constants; others don’t.

Did we mean Language One to do this?
Value Of A Sum

?- val(plus(const(2147483647),const(1)),X).
X = 2.14748e+009.

?- val(plus(const(2147483647),const(1)),X).
X = 2147483648.

Some Prologs expresses sums greater than $2^{31}-1$ as floating-point results; others don’t.

Did we mean Language One to do this?
Defining Semantics By Interpreter

- Our `val1` is not satisfactory as a definition of the semantics of Language One
- “Language One programs behave the way this interpreter says they behave, *running under this implementation of Prolog on this computer system*”
- We need something more abstract
Natural Semantics

A formal notation we can use to capture the same basic proof rules in val1

We are trying to define the relation between an AST and the result of evaluating it

We will use the symbol → for this relation, writing \( E \rightarrow v \) to mean that the AST \( E \) evaluates to the value \( v \)

For example, our semantics should establish \( \text{times}(\text{const}(2),\text{const}(3)) \rightarrow 6 \)
A Rule In Natural Semantics

\[
\frac{E_1 \rightarrow \nu_1 \quad E_2 \rightarrow \nu_2}{\text{times}(E_1, E_2) \rightarrow \nu_1 \times \nu_2}
\]

- Conditions above the line, conclusion below
- The same idea as our Prolog rule:

```prolog
val1(times(X,Y),Value) :-
    val1(X,XValue),
    val1(Y,YValue),
    Value is XValue * YValue.
```
Language One, Natural Semantics

\[
\begin{align*}
\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\text{plus}(E_1,E_2) \rightarrow v_1 + v_2} \\
\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\text{times}(E_1,E_2) \rightarrow v_1 \times v_2} \\
\text{const}(n) \rightarrow \text{eval}(n)
\end{align*}
\]

\[\text{val1}(\text{plus}(X,Y),\text{Value}) : - \\
\text{val1}(X,\text{XValue}) , \\
\text{val1}(Y,\text{YValue}) , \\
\text{Value is XValue + YValue} .
\]

\[\text{val1}(\text{times}(X,Y),\text{Value}) : - \\
\text{val1}(X,\text{XValue}) , \\
\text{val1}(Y,\text{YValue}) , \\
\text{Value is XValue * YValue} .
\]

\[\text{val1}(\text{const}(X),X) .\]

Of course, this still needs definitions for +, × and \textit{eval}, but at least it won’t accidentally use Prolog’s
Natural Semantics, Note

- There may be more than one rule for a particular kind of AST node
- For instance, for an ML-style if-then-else we might use something like this:

\[
\begin{align*}
E_1 \rightarrow \text{true} & \quad E_2 \rightarrow v_2 \\
\text{if}(E_1, E_2, E_3) \rightarrow v_2
\end{align*}
\]

\[
\begin{align*}
E_1 \rightarrow \text{false} & \quad E_3 \rightarrow v_3 \\
\text{if}(E_1, E_2, E_3) \rightarrow v_3
\end{align*}
\]
Outline

- Natural semantics and Prolog interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions
Defining Language Two

- That one was too easy!
- To make it a little harder, let’s add:
  - Variables
  - An ML-style `let` expression for defining them
Syntax

\[ <\text{exp}> ::= <\text{exp}> + <\text{mulexp}> | <\text{mulexp}> \]
\[ <\text{mulexp}> ::= <\text{mulexp}> * <\text{rootexp}> | <\text{rootexp}> \]
\[ <\text{rootexp}> ::= \text{let val} <\text{variable}> = <\text{exp}> \text{ in } <\text{exp}> \text{ end} \]
\[ | ( <\text{exp}> ) | <\text{variable}> | <\text{constant}> \]

(A subset of ML expressions)

This grammar is unambiguous

A sample Language Two expression:
\[
\text{let val } y = 3 \text{ in } y*y \text{ end}
\]
Abstract Syntax

- Two more kinds of AST nodes:
  - `var(X)` for a reference to a variable `X`
  - `let(X,Exp1,Exp2)` for a `let` expression that evaluates `Exp2` in an environment where the variable `X` is bound to the value of `Exp1`

- So for the Language Two program
  ```
  let val y = 3 in y*y end
  ```
  we have this AST:
  ```
  let(y,const(3),times(var(y),var(y)))
  ```
Representing Contexts

- A representation for contexts:
  - $\text{bind}(\text{Variable}, \text{Value}) = \text{the binding from Variable to Value}$
  - A context is a list of zero or more $\text{bind}$ terms

- For example:
  - The context in which $\text{y}$ is bound to 3 could be $[\text{bind}(\text{y}, 3)]$
  - The context in which both $\text{x}$ and $\text{y}$ are bound to 3 could be $[\text{bind}(\text{x}, 3), \text{bind}(\text{y}, 3)]$
Looking Up A Binding

lookup(Variable, [bind(Variable,Value)|_], Value) :- !.
lookup(VarX, [__|Rest], Value) :-
    lookup(VarX, Rest, Value).

- Looks up a binding in a context
- Finds the most recent binding for a given variable, if more than one
Language Two: Prolog Interpreter

\texttt{val2(plus(X,Y),Context,Value) :-}
   \texttt{val2(X,Context,XValue),}
   \texttt{val2(Y,Context,YValue),}
   \texttt{Value is XValue + YValue.}
\texttt{val2(times(X,Y),Context,Value) :-}
   \texttt{val2(X,Context,XValue),}
   \texttt{val2(Y,Context,YValue),}
   \texttt{Value is XValue * YValue.}
\texttt{val2(const(X),_,X).}
\texttt{val2(var(X),Context,Value) :-}
   \texttt{lookup(X,Context,Value).}
\texttt{val2(let(X,Exp1,Exp2),Context,Value2) :-}
   \texttt{val2(Exp1,Context,Value1),}
   \texttt{val2(Exp2,[bind(X,Value1)|Context],Value2).}
?- val2(let(y,const(3),times(var(y),var(y))),nil,X).
X = 9.

let val y = 3 in y*y end
?- val2(let(y, const(3),
  |    let(x, times(var(y), var(y)),
  |      times(var(x), var(x))),
  |    nil, X).
X = 81.

let val y = 3 in
  let val x = y*y in
    x*x
  end
end
?- val2(let(y,const(1),let(y,const(2),var(y))),nil,X).
X = 2.

let val y = 1 in
  let val y = 2 in
    y
  end
end
Natural Semantics

- As before, we will write a natural semantics to capture the same basic proof rules.
- We will again use the symbol $\rightarrow$ for this relation, though it is a different relation.
- We will write $<E, C> \rightarrow v$ to mean that the value of the AST $E$ in context $C$ is $v$. 
Language Two, Natural Semantics

\[
\begin{align*}
\frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2}{\langle \text{plus}(E_1, E_2), C \rangle \rightarrow v_1 + v_2} & \quad \langle \text{var}(v), C \rangle \rightarrow \text{lookup}(C, v) \\
\frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2}{\langle \text{times}(E_1, E_2), C \rangle \rightarrow v_1 \times v_2} & \quad \langle \text{const}(n), C \rangle \rightarrow \text{eval}(n) \\
\frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, \text{bind}(x, v_1) :: C \rangle \rightarrow v_2}{\langle \text{let}(x, E_1, E_2), C \rangle \rightarrow v_2}
\end{align*}
\]

This still needs definitions for $+$, $\times$ and $\text{eval}$, as well as $\text{bind}$, $\text{lookup}$, $::$, and the nil environment
About Errors

■ In Language One, all syntactically correct programs run without error
■ Not true in Language Two:
  \[
  \text{let val a = 1 in b end}
  \]
■ What does the semantics say about this?
Undefined Variable Error

?- val2(let(a,const(1),var(b)),nil,X).
   false.

Our natural semantics says something similar: there is no $v$ for which

$$\langle \text{let}(a,\text{const}(1),\text{var}(b)), \text{nil} \rangle \rightarrow v$$
Static Semantics

■ Ordinarily, language systems perform error checks after parsing but before running
  – For static scoping: references must be in the scope of some definition of the variable
  – For static typing: a consistent way to assign a type to every part of the program

■ This part of a language definition, neither syntax nor runtime behavior, is called static semantics
Static and Dynamic Semantics

Language Two semantics could be 2 parts:
- Static semantics rules out runtime errors
- Dynamic semantics can ignore the issue

Static semantics can be complicated too:
- ML’s type inference
- Java’s “definite assignment”

In this chapter, dynamic semantics only
Note: Dynamic Error Semantics

- In full-size languages, there are still things that can go wrong at runtime
- One approach is to define error outcomes in the natural semantics:
  \[
  \langle \text{divide(const(6), const(3)), } C \rangle \rightarrow \langle \text{normal, 2} \rangle
  \]
  \[
  \langle \text{divide(const(6), const(0)), } C \rangle \rightarrow \langle \text{abrupt, zerodivide} \rangle
  \]
- Today: semantics for error-free case only
Outline

Natural semantics and Prolog interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions
Defining Language Three

To make it a little harder, let’s add:
- ML-style function values
- ML-style function application
Syntax

\[
\begin{align*}
\text{<exp>} & : = \textbf{fn} \ <\text{variable}> \ =\> \ <\text{exp}> \ | \ <\text{addexp}> \\
\text{<addexp>} & : = \ <\text{addexp}> \ + \ <\text{mulexp}> \ | \ <\text{mulexp}> \\
\text{<mulexp>} & : = <\text{mulexp}> \ * \ <\text{funexp}> \ | \ <\text{funexp}> \\
\text{<funexp>} & : = <\text{funexp}> \ <\text{rootexp}> \ | \ <\text{rootexp}> \\
\text{<rootexp>} & : = \textbf{let} \ \textbf{val} \ <\text{variable}> \ = \ <\text{exp}> \ \textbf{in} \ <\text{exp}> \ \textbf{end} \\
& \hspace{1cm} | \ (<\text{exp}>)) \ | \ <\text{variable}> \ | \ <\text{constant}> \\
\end{align*}
\]

- (A subset of ML expressions)
- This grammar is unambiguous
- Function application has highest precedence
- A sample Language Three expression:
  \[(\text{fn} \ x \ =\> \ x \ * \ x) \ 3\]
Abstract Syntax

Two more kinds of AST nodes:

- `apply(Function,Actual)` applies the `Function` to the `Actual` parameter
- `fn(Formal,Body)` for an fn expression with the given `Formal` parameter and `Body`

So for the Language Three program

\[(fn \ x \Rightarrow \ x \times \ x) \ 3\]

we have this AST:

\[
\text{apply}(\text{fn}(x,\text{times}(\text{var}(x),\text{var}(x))), \text{const}(3))
\]
Representing Functions

- A representation for functions:
  - `fval(Formal,Body)`
  - `Formal` is the formal parameter variable
  - `Body` is the unevaluated function body

- So the AST node `fn(Formal,Body)` evaluates to `fval(Formal,Body)`

- (Why not just use the AST node itself to represent the function? You’ll see…)
Language Three: Prolog Interpreter

val3(plus(X,Y),Context,Value) :- ...
val3(times(X,Y),Context,Value) :- ...
val3(const(X),_,X).
val3(var(X),Context,Value) :- ...
val3(let(X,Exp1,Exp2),Context,Value2) :- ...

Same as for Language Two

val3(fn(Formal,Body),_,fval(Formal,Body)).
val3(apply(Function,Actual),Context,Value) :-
  val3(Function,Context,fval(Formal,Body)),
  val3(Actual,Context,ParamValue),
  val3(Body,[bind(Formal,ParamValue)|Context],Value).
?- val3(apply(fn(x,times(var(x),var(x)))),
|       const(3)),
|       nil,X).
\nX = 9.

\n(fn x => x * x) 3
Question

What should the value of this Language Three program be?

```ml
let val x = 1 in
  let val f = fn n => n + x in
    let val x = 2 in
      f 0
    end
  end
end
```

- Depends on whether scoping is static or dynamic
?- val3(let(x,const(1),
|       let(f,fn(n,plus(var(n),var(x)))),
|       let(x,const(2),
|           apply(var(f),const(0)))),
|       nil,X).

x = 2.

let val x = 1 in
  let val f = fn n => n + x in
    let val x = 2 in
      f 0
    end
  end
end

Oops—we defined Language Three with dynamic scoping!
Dynamic Scoping

- We got dynamic scoping
- Probably not a good idea:
  - We have seen its drawbacks: difficult to implement efficiently, makes large complex scopes
  - Most modern languages use static scoping
- How can we fix this so that Language Three uses static scoping?
Representing Functions, Again

- Add context to function representation:
  - \( \text{fval}(\text{Formal}, \text{Body}, \text{Context}) \)
  - \text{Formal} is the formal parameter variable
  - \text{Body} is the unevaluated function body
  - \text{Context} is the context to use when calling it

- So the AST node \( \text{fn}(\text{Formal}, \text{Body}) \)
  evaluated in \text{Context}, produces to
  \( \text{fval}(\text{Formal}, \text{Body}, \text{Context}) \)

- \text{Context} works as a \textit{nesting link} (Chapter 12)
Language Three: Prolog Interpreter, Static Scoping

val3(fn(Formal,Body),_,fval(Formal,Body)).

\[\downarrow\]

val3(fn(Formal,Body),Context,fval(Formal,Body,Context)).

\[\downarrow\]

val3(apply(Function,Actual),Context,Value) :-
    val3(Function,Context,fval(Formal,Body)),
    val3(Actual,Context,ParamValue),
    val3(Body,bind(Formal,ParamValue,Context),Value).

\[\downarrow\]

val3(apply(Function,Actual),Context,Value) :-
    val3(Function,Context,fval(Formal,Body,Nesting)),
    val3(Actual,Context,ParamValue),
    val3(Body,[bind(Formal,ParamValue)|Nesting],Value).
?- val3(let(x,const(1),
|   let(f,fn(n,plus(var(n),var(x)))),
|   let(x,const(2),
|       apply(var(f),const(0)))),
|   nil,X).
X = 1.

let val x = 1 in
  let val f = fn n => n + x in
    let val x = 2 in
      f 0
    end
  end
end

That’s better: static scoping!
?- val3(let(f,fn(x,let(g,fn(y,plus(var(y),var(x)))),
|                   var(g))),
|             apply(apply(var(f),const(1)),const(2))),
|            nil,X).

X = 3.

let
  val f = fn x =>
    let val g = fn y => y+x in
    g
  end
in
  f 1 2
end

Handles ML-style higher order functions.
Language Three Natural Semantics, Dynamic Scoping

\[
\begin{align*}
\langle E_1, C \rangle & \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2 \\
\langle \text{plus}(E_1, E_2), C \rangle & \rightarrow v_1 + v_2 \\
\langle E_1, C \rangle & \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2 \\
\langle \text{times}(E_1, E_2), C \rangle & \rightarrow v_1 \times v_2 \\
\langle E_1, C \rangle & \rightarrow v_1 \quad \langle E_2, \text{bind}(x, v_1) :: C \rangle \rightarrow v_2 \\
\langle \text{let}(x, E_1, E_2), C \rangle & \rightarrow v_2 \\
\langle E_1, C \rangle & \rightarrow (x, E_3) \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, \text{bind}(x, v_1) :: C \rangle \rightarrow v_2 \\
\langle \text{apply}(E_1, E_2), C \rangle & \rightarrow v_2
\end{align*}
\]
Language Three Natural Semantics, Static Scoping

\[ \langle \text{fn}(x, E), C \rangle \rightarrow (x, E) \]
\[ \downarrow \]
\[ \langle \text{fn}(x, E), C \rangle \rightarrow (x, E, C) \]

\[ \langle E_1, C \rangle \rightarrow (x, E_3) \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, \text{bind}(x, v_1) :: C \rangle \rightarrow v_2 \]
\[ \frac{}{\langle \text{apply}(E_1, E_2), C \rangle \rightarrow v_2} \]
\[ \downarrow \]
\[ \langle E_1, C \rangle \rightarrow (x, E_3, C') \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, \text{bind}(x, v_1) :: C' \rangle \rightarrow v_2 \]
\[ \frac{}{\langle \text{apply}(E_1, E_2), C \rangle \rightarrow v_2} \]
About Errors

Language Three now has more than one type, so we can have type errors:

```
?- val3(apply(const(1),const(1)),nil,X).
false.
```

Similarly, the natural semantics gives no \( \nu \) for which

\[
<\text{apply}(\text{const}(1),\text{const}(1)), \text{nil}> \rightarrow \nu
\]
More Errors

- In the dynamic-scoping version, we can also have programs that run forever:

  ```ml
  let val f = fn x => f x in f 1 end
  ```

- Interpreter runs forever on this

- Natural semantics does not run forever—does not *run* at all—it just defines no result for the program
Outline

- Natural semantics and Prolog interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions

- Natural semantics is one of many formal techniques for defining semantics

- Other techniques: see the last section of the chapter for a summary