

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 (ASTs)
- Now... the rest of the story: formal definitions of programming language semantics

Kinds Of Formal Semantics

- There are many kinds, including:
 - Operational
 - Big-step: *natural semantics*
 - Small-step: *structural operational semantics (SOS)*
 - Abstract machine
 - Axiomatic
 - Denotational
- Today: natural semantics, because it is related to what we've been doing with Prolog

Outline

- 23.2 Language One
- 23.3 Language Two—Adding Variables
- 23.4 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

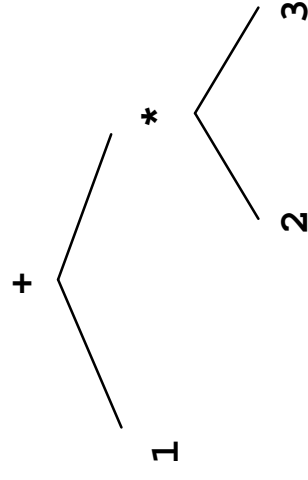
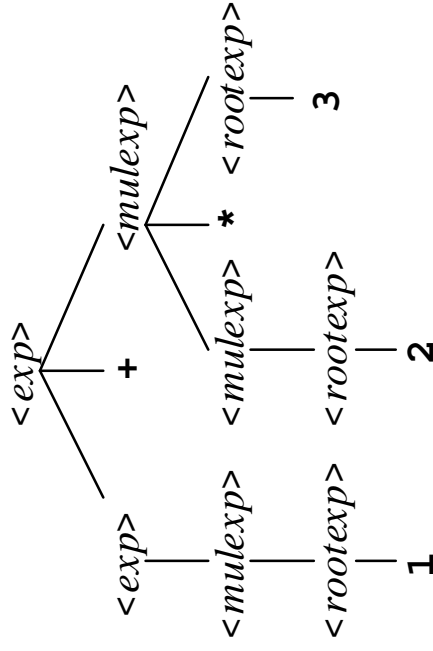
Syntax: Phrase Structure

$$\begin{aligned} \langle exp \rangle & ::= \langle exp \rangle + \langle mulexp \rangle \mid \langle mulexp \rangle \\ \langle mulexp \rangle & ::= \langle mulexp \rangle * \langle rootexp \rangle \mid \langle rootexp \rangle \\ \langle rootexp \rangle & ::= (\langle exp \rangle) \mid \langle constant \rangle \end{aligned}$$

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

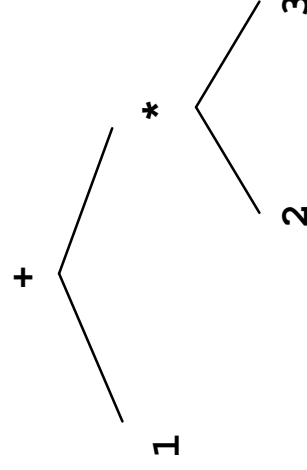
Parse Trees And ASTs

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



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 ASTs as input:



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

Abstract Syntax

- Note: the set of legal ASTs can be defined by a grammar:

```
<ast> ::= plus( <ast> , <ast> )  
        | times( <ast> , <ast> )  
        | const( <constant> )
```

- This is called an *abstract syntax* for the language
- The original grammar was the *concrete syntax* of the language

Language One: Prolog Interpreter

```
    val1(plus(X, Y), Value) :-  
        val1(X, XValue),  
        val1(Y, YValue),  
        Value is XValue + YValue.  
    val1(times(X, Y), Value) :-  
        val1(X, XValue),  
        val1(Y, YValue),  
        Value is XValue * YValue.  
    val1(const(X), X).
```

```
?- val1(const(1),X).
```

```
X = 1
```

```
Yes
```

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

```
X = 3
```

```
Yes
```

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

```
X = 7
```

```
Yes
```

Problems

- What is the value of a constant?
 - Interpreter says `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

```
?- val1(const(2147483647),X).
```

```
X = 2147483647
```

Yes

```
?- val1(const(2147483648),X).
```

```
X = 2.14748e+009
```

Yes

- Our Prolog treats values greater than $2^{31}-1$ as floating-point constants
- Did we mean Language One to do this?

Value Of A Sum

```
?- val1(plus(const(2147483646),const(1)),X).
```

```
X = 2147483647
```

Yes

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

```
X = 2.14748e+009
```

Yes

- Our Prolog expresses sums greater than $2^{31}-1$ as floating-point results
- 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 \rightarrow for this relation, writing $E \rightarrow v$ to mean that the AST E evaluates to the value v
- For example, our semantics should establish **times (const (2) , const (3)) \rightarrow 6**

A Rule In Natural Semantics

$$\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\mathbf{times}(E_1, E_2) \rightarrow v_1 \times v_2}$$

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

```
val1(times(X, Y), Value) :-  
    val1(X, XValue),  
    val1(Y, YValue),  
    Value is XValue * YValue.
```

Language One, Natural Semantics

$$\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\text{plus}(E_1, E_2) \rightarrow v_1 + v_2}$$

$$\text{plus}(E_1, E_2) \rightarrow v_1 + v_2$$

$$\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\text{times}(E_1, E_2) \rightarrow v_1 \times v_2}$$

$$\text{times}(E_1, E_2) \rightarrow v_1 \times v_2$$

$$\text{const}(n) \rightarrow \text{eval}(n)$$

```
val1(plus(X, Y), Value) :-  
    val1(X, XValue),  
    val1(Y, YValue),  
    Value is XValue + YValue.  
val1(times(X, Y), Value) :-  
    val1(X, XValue),  
    val1(Y, YValue),  
    Value is XValue * YValue.  
val1(const(X), X).
```

- Of course, this still needs definitions for $+$, \times and *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:

$$\frac{E_1 \rightarrow true \quad E_2 \rightarrow v_2}{\mathbf{if}(E_1, E_2, E_3) \rightarrow v_2}$$

$$\frac{E_1 \rightarrow false \quad E_3 \rightarrow v_3}{\mathbf{if}(E_1, E_2, E_3) \rightarrow v_3}$$

Outline

- 23.2 Language One
- 23.3 Language Two—Adding Variables
- 23.4 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

$\langle exp \rangle ::= \langle exp \rangle + \langle mulexp \rangle \mid \langle mulexp \rangle$
 $\langle mulexp \rangle ::= \langle mulexp \rangle * \langle rootexp \rangle \mid \langle rootexp \rangle$
 $\langle rootexp \rangle ::= \text{let } \mathbf{val} \langle variable \rangle = \langle exp \rangle \text{ in } \langle exp \rangle \text{ end}$
 $\mid (\langle exp \rangle) \mid \langle variable \rangle \mid \langle constant \rangle$

- (A subset of ML expressions)
- This grammar is unambiguous
- A sample Language Two expression:
let val y = 3 in y*y 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:
 - `bind(Variable, Value)` = the binding from `Variable` to `Value`
 - A context is a list of zero or more `bind` terms
- For example:
 - The context in which `y` is bound to 3 could be `[bind(y, 3)]`
 - The context in which both `x` and `y` are bound to 3 could be `[bind(x, 3), bind(y, 3)]`

Looking Up A Binding

```
lookup(Variable, [bind(Variable, Value) | _], Value) .  
lookup(VarX, [bind(VarY, _) | Rest], Value) :-  
    VarX \= VarY,  
    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

```
val2(plus(X, Y), Context, Value) :-  
    val2(X, Context, XValue),  
    val2(Y, Context, YValue),  
    Value is XValue + YValue.  
val2(times(X, Y), Context, Value) :-  
    val2(X, Context, XValue),  
    val2(Y, Context, YValue),  
    Value is XValue * YValue.  
val2(const(X), _, X).  
val2(var(X), Context, Value) :-  
    lookup(X, Context, Value).  
val2(let(X, Exp1, Exp2), Context, Value2) :-  
    val2(Exp1, Context, Value1),  
    val2(Exp2, [bind(X, Value1) | Context], Value2).
```

```
?- val2(let(y, const(3), times(var(y), var(y))), nil, x) .
```

```
X = 9
```

```
Yes
```

```
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

Yes

```
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
```

```
Yes
```

```
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 $\langle E, C \rangle \rightarrow v$ to mean that the value of the AST E in context C is v

Language Two, Natural Semantics

$$\frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2}{\langle \mathbf{plus}(E_1, E_2), C \rangle \rightarrow v_1 + v_2}$$
$$\frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2}{\langle \mathbf{times}(E_1, E_2), C \rangle \rightarrow v_1 \times v_2}$$
$$\langle \mathbf{var}(v), C \rangle \rightarrow \mathit{lookup}(C, v)$$
$$\langle \mathbf{const}(n), C \rangle \rightarrow \mathit{eval}(n)$$
$$\frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, \mathbf{bind}(x, v_1) :: C \rangle \rightarrow v_2}{\langle \mathbf{let}(x, E_1, E_2), C \rangle \rightarrow v_2}$$

- This still needs definitions for $+$, \times and eval , as well as bind , lookup , $::$, and the nil environment

About Errors

- In Language One, all syntactically correct programs run without error
- Not true in Language Two:
let val a = 1 in b end
- What does the semantics say about this?

Undefined Variable Error

```
?- val12(let(a, const(1), var(b)), nil, x).
```

No

- 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:

$\langle \text{divide}(\text{const}(6), \text{const}(3)), C \rangle \rightarrow \langle \text{normal}, 2 \rangle$

$\langle \text{divide}(\text{const}(6), \text{const}(0)), C \rangle \rightarrow \langle \text{abrupt}, \text{zerodivide} \rangle$

- Today: semantics for error-free case only

Outline

- 23.2 Language One
- 23.3 Language Two—Adding Variables
- 23.4 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

$\langle exp \rangle ::= \mathbf{fn} \langle variable \rangle = \langle exp \rangle \mid \langle addexp \rangle$
 $\langle addexp \rangle ::= \langle addexp \rangle + \langle mulexp \rangle \mid \langle mulexp \rangle$
 $\langle mulexp \rangle ::= \langle mulexp \rangle * \langle funexp \rangle \mid \langle funexp \rangle$
 $\langle funexp \rangle ::= \langle funexp \rangle \langle rootexp \rangle \mid \langle rootexp \rangle$
 $\langle rootexp \rangle ::= \mathbf{let} \mathbf{val} \langle variable \rangle = \langle exp \rangle \mathbf{in} \langle exp \rangle \mathbf{end}$
 $\mid (\langle exp \rangle) \mid \langle variable \rangle \mid \langle constant \rangle$

- (A subset of ML expressions)
- This grammar is unambiguous
- Function application has highest precedence
- A sample Language Three expression:

(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 => x * x) 3
```

we have this AST:

```
apply(fn(x, times(var(x), var(x))),  
      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) :- ...   Same as for
val3(const(x), _, x) .                    Language Two
val3(var(x), Context, Value) :- ...
val3(let(x, Exp1, Exp2), Context, Value2) :- ...
```

```
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).
```

X = 9

Yes

```
(fn x => x * x) 3
```

Question

- What should the value of this Language Three program be?

```
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

Yes

```
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:
 - **fval(Formal, Body, Context)**
 - **Formal** is the formal parameter variable
 - **Body** is the unevaluated function body
 - **Context** is the context to use when calling it
- So the AST node **fn(Formal, Body)** evaluated in **Context**, produces to **fval(Formal, Body, Context)**
- **Context** works as a *nesting link* (Chapter 12)

Language Three: Prolog Interpreter, Static Scoping

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



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

```
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(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

Yes

```
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

Yes

```
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

$$\begin{array}{c}
 \frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2}{\langle \mathbf{plus}(E_1, E_2), C \rangle \rightarrow v_1 + v_2} \\
 \frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2}{\langle \mathbf{times}(E_1, E_2), C \rangle \rightarrow v_1 \times v_2} \\
 \langle \mathbf{const}(n), C \rangle \rightarrow \mathit{eval}(n) \\
 \langle \mathbf{var}(v), C \rangle \rightarrow \mathit{lookup}(C, v) \\
 \langle \mathbf{fn}(x, E), C \rangle \rightarrow (x, E) \\
 \frac{\langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, \mathbf{bind}(x, v_1) :: C \rangle \rightarrow v_2}{\langle \mathbf{let}(x, E_1, E_2), C \rangle \rightarrow v_2} \\
 \frac{\langle E_1, C \rangle \rightarrow (x, E_3) \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, \mathbf{bind}(x, v_1) :: C \rangle \rightarrow v_2}{\langle \mathbf{apply}(E_1, E_2), C \rangle \rightarrow v_2}
 \end{array}$$

Language Three Natural Semantics, Static Scoping

$$\langle \mathbf{fn}(x, E), C \rangle \rightarrow (x, E)$$

\Downarrow

$$\langle \mathbf{fn}(x, E), C \rangle \rightarrow (x, E, C)$$

$$\frac{\langle E_1, C \rangle \rightarrow (x, E_3) \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, \mathit{bind}(x, v_1) :: C \rangle \rightarrow v_2}{\langle \mathbf{apply}(E_1, E_2), C \rangle \rightarrow v_2}$$

\Downarrow

$$\frac{\langle E_1, C \rangle \rightarrow (x, E_3, C') \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, \mathit{bind}(x, v_1) :: C' \rangle \rightarrow v_2}{\langle \mathbf{apply}(E_1, E_2), C \rangle \rightarrow v_2}$$

About Errors

- Language Three now has more than one type, so we can have type errors: **1**

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

No

- Similarly, the natural semantics gives no v for which

$\langle \text{apply}(\text{const}(1), \text{const}(1)), \text{nil} \rangle \rightarrow v$

More Errors

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

```
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

Conclusion

- Today's examples used natural semantics
- For other kinds, see the last section of the chapter and the Further Reading
- Uses for formal semantics:
 - Defining languages for implementation
 - Justifying program transformations
 - Proving programs correct
- Plus: *they are very hard to read!*

Science is a way of trying not to fool yourself.
— Richard Feynman