COMP 356
Programming Language Structures
Notes for Chapter 3 of *Concepts of Programming Languages*
Syntax and Semantics

Definitions:

- **syntax** is the form (structure, grammar) of a language
- **semantics** is the meaning of a language

Example:

```
if (a > b) a = a + 1;
else b = b + 1;
```

- syntax: `if-else` is an operator that takes three operands - a condition and two statements
- semantics: if the value of `a` is greater than the value of `b`, then increment `a`. Otherwise, increment `b`.

Both the syntax and semantics of a programming language must be carefully defined so that:

- language implementors can implement the language (correctly), so that programs developed with one implementation run correctly under another (portability)
- programmers can use the language (correctly)

1 Describing Syntax

A **language** is a set of strings of characters from some alphabet.
Examples:

- English (using the standard alphabet)
- binary numbers (using the alphabet $\{0, 1\}$)

The syntax rules of a language determine whether or not arbitrary strings belong to the language. The first step in specifying syntax is describing the basic units or "words" of the language, called **lexemes**. For example, some typical Java lexemes include:

- `if`
- `++`
- `+`
- `3.27`
- `count`

Lexemes are grouped into categories called **tokens**. Each token has one or more lexemes.
Tokens are specified using regular expressions or finite automata.
The scanner/lexical analyzer of a compiler processes the character strings in the source program and determines the tokens that they represent.
Once the tokens of a language are defined, the next step is to determine which sequences of tokens are in the language (syntax).

token	**sample lexemes**
identifier	`count`, `i`, `x2`, ...
if	`if`
add_op	`+`, `-`
semi	`;`
int_lit	`0`, `10`, `-2.4`

Table 1: Example tokens and some associated lexemes.

1.1 Using BNF to Describe Syntax

BNF ≡ Backus-Naur Form

BNF is:

- a **metalanguage** - a language used to describe other languages
- the standard way to describe programming language syntax
- often used in language reference manuals

The class (set) of languages that can be described using BNF is called the **context-free languages**, and BNF descriptions are also called **context-free grammars** or just **grammars**.

symbol	**meaning**
→	is defined as
|	or (alternatives)
<something>	a **nonterminal** - replace by the definition of something
something	(with no < >) a **token** or **terminal** - a "word" in the language being defined (not replaced)
ϵ	the empty string (nothing)

Table 2: BNF notation.

Example (if statements in C):

<if-stmt> → if (<expr>) <stmt>
<if-stmt> → if (<expr>) <stmt> else <stmt>

Each definition above is called a **rule** or **production**. A full BNF definition would include definitions for all the nonterminals used - <expr> and <stmt> are not defined above.

The two rules above can be abbreviated using | as follows:

<if-stmt> → if (<expr>) <stmt>
    | if (<expr>) <stmt> else <stmt>

Example (expressions in C):

<expr> → id | num | (<expr>) | <expr> <op> <expr>
<op> → `+` | `-` | `*` | `/` | `==` | `<` | `<=` | `>` | `>=`

Note that BNF definitions can be recursive.

1.2 Derivations

A **derivation** is a sequence of replacements using the rules of a grammar. Derivations:

- are often used to show that a particular sequence of tokens belongs to the language defined by the grammar

- always begin with the **start symbol** for the grammar
  - by tradition, the nonterminal on the LHS of the first rule is the start symbol
  - o.w. the start symbol could have a special name such as <start> or <program>

Example - a derivation to show that:

id + num

is a valid expression (<expr>):

```
<expr> ⇒ <expr> <op> <expr>
       ⇒ id <op> <expr>
       ⇒ id + <expr>
       ⇒ id + num
```

Example - a derivation to show that:

id < (num / num)

is a valid expression (<expr>):

```
<expr> ⇒ <expr> <op> <expr>
       ⇒ id <op> <expr>
       ⇒ id < <expr>
       ⇒ id < (<expr>)
       ⇒ id < (<expr> <op> <expr>)
       ⇒ id < (num <op> <expr>)
       ⇒ id < (num / <expr>)
       ⇒ id < (num / num)
```

Definitions:

- the symbol ⇒ is called **derives**
- each string of symbols derived from the start symbol (including the start symbol itself) is called a **sentential form**
- a **leftmost derivation** is a derivation in which the leftmost nonterminal is always chosen for replacement
- a **rightmost derivation** is a derivation in which the rightmost nonterminal is always chosen for replacement

Derivation order has no affect on the set of strings that can be derived.
BNF example - a subset of statements in C:

```
<stmt> → <if-stmt> | <loop-stmt> | <assign-stmt> | <cmpd-stmt>
<if-stmt> → if (<expr>) <stmt>
          | if (<expr>) <stmt> else <stmt>
<loop-stmt> → while (<expr>) <stmt>
<assign-stmt> → id = <expr> ;
<cmpd-stmt> → { <stmt-list> }
<stmt-list> → ϵ | <stmt> | <stmt> <stmt-list>
<expr> → id | num | (<expr>) | <expr> <op> <expr>
<op> → + | - | * | / | == | < | <= | > | >=
```

Even more definitions:

- a BNF rule in which the nonterminal on the LHS is also the first (leftmost) symbol on the RHS is **left recursive**

- a BNF rule in which the nonterminal on the LHS is also the last (rightmost) symbol on the RHS is **right recursive**

For example:

- <stmt-list> → <stmt> <stmt-list>
  is right recursive
- <expr> → <expr> <op> <expr>
  is both left and right recursive

1.3 Parse Trees

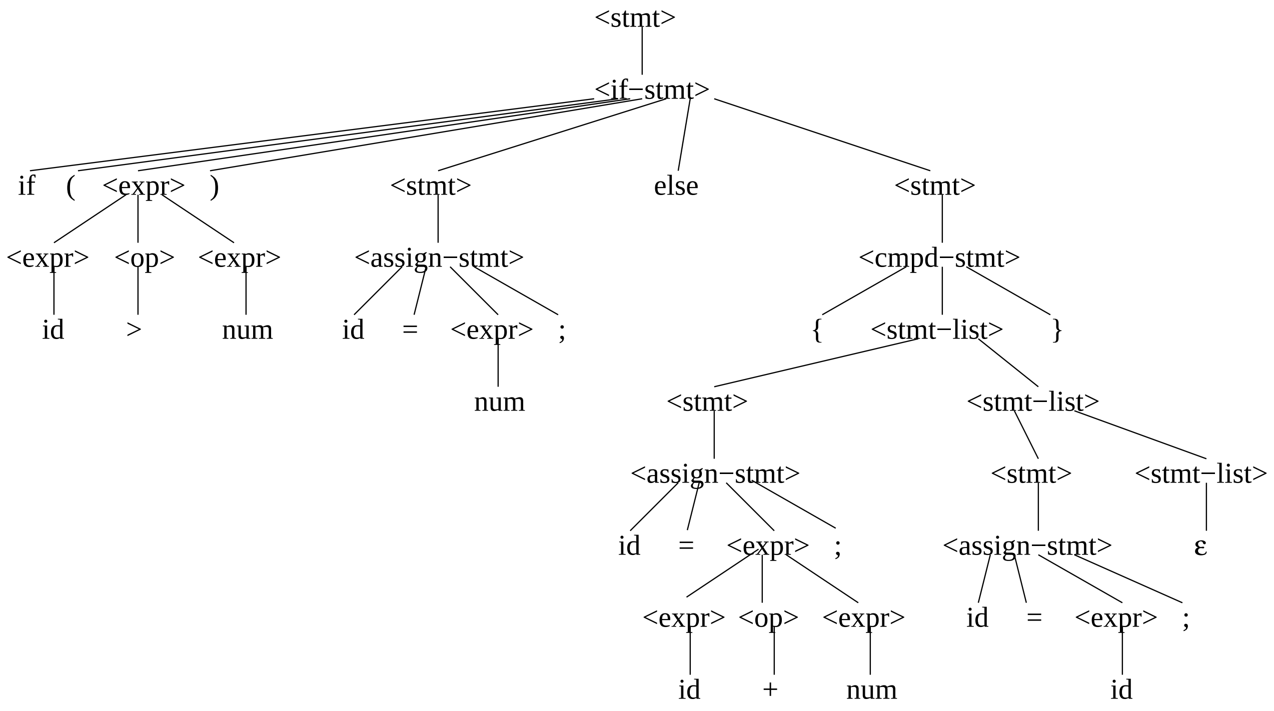
A **parse tree** is a graphical way of representing a derivation.

- the root of the parse tree is always the start symbol
- each interior node is a nonterminal
- each leaf node is a token
- the children of a nonterminal (interior node) are the RHS of some rule whose LHS is the nonterminal

For example, a parse tree for:

if (id > num) id = num; else { id = id + num; id = id; }

using the previous grammar is:



Notes on parse trees:

- the parser phase of a compiler organizes tokens into parse trees
- parse trees explicitly show the structure and order of evaluation of language constructs
- there is a one-to-one correspondence between parse trees and leftmost derivations for a string

A grammar is **ambiguous** if there are 2 or more distinct parse trees (or equivalently, leftmost derivations) for the same string.

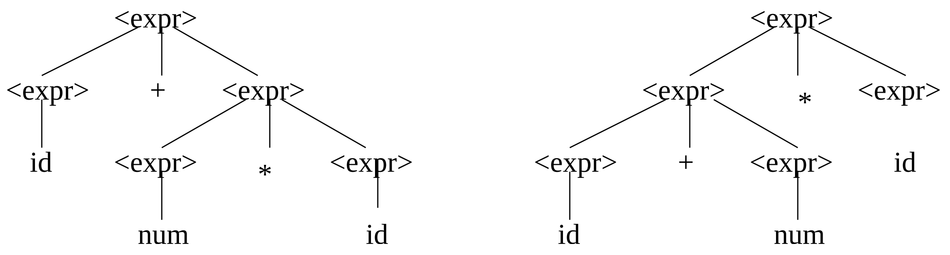
Consider the grammar:

<expr> → id | num | (<expr>) | <expr> + <expr> | <expr> * <expr>

and the string:

id + num * id

The following parse trees show that this grammar is ambiguous:



Which parse tree would we prefer?

Grammars can often be modified to remove ambiguity - in this case, by enforcing associativity and precedence:

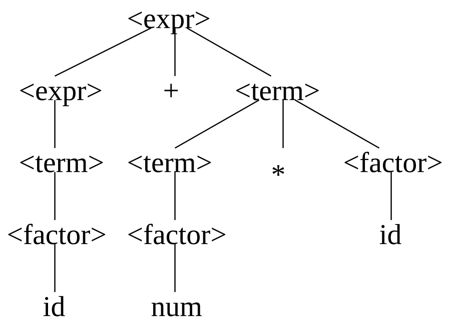
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → id | num | (<expr>)

The intuition behind this approach is to try to force the + to occur higher in the parse tree.

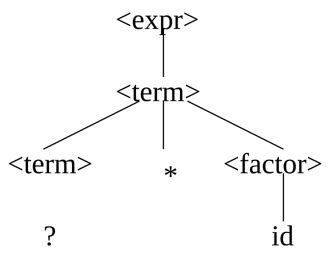
The parse tree for :

id + num * id

is:



Trying to find another tree:



This grammar modification gives rise to three proof obligations:

- the two grammars define the same language
- the second grammar always gives correct associativity and precedence
- the second grammar is not ambiguous

These proofs are omitted.

1.4 Extended BNF (EBNF)

EBNF provides some additional notation, but does not allow any additional languages to be defined (as compared to BNF). Hence, it is only a notational convenience.

notation	**meaning**
[x]	optional (0 or 1 occurrence of x)
{x}	repetition (0 or more occurrences of x)
(x | y)	choice - x or y

Table 3: Additional EBNF notation.

Examples:

BNF: <stmt-list> → ϵ | <stmt> <stmt-list>
EBNF: <stmt-list> → {<stmt>}

BNF: <expr> → <expr> + <expr> | <expr> * <expr>
EBNF: <expr> → <expr> (+ | *) <expr>

BNF: <expr> → <expr> + <term> | <term>
EBNF: <expr> → [<expr> +] <term>

2 Static Semantics and Attribute Grammars

Static semantics is any aspect of semantics (meaning) that can be checked at compile time (statically). Examples:

- checking that variables are declared before they are used
- type checking

Many language properties (including those mentioned above) can be checked statically, but are impossible or difficult to specify with a BNF definition.

An **attribute grammar** is a grammar with attributes (properties) attached to each symbol, plus rules for computing and checking the values of attributes. Attribute grammars are useful for checking static semantic properties.

If parse trees are implemented with a record or class instance for each node, then the attribute is just an additional field in this representation.

Types of attributes:

- a **synthesized attribute** is an attribute whose value is computed based on the attribute values of child nodes (in a parse tree). Synthesized attributes pass information up the parse tree.
- an **inherited attribute** is an attribute whose value is computed based on the attribute values of parent and sibling nodes (in a parse tree). Synthesized attributes pass information down the parse tree.

- an **intrinsic attribute** is an attribute whose value is not based on other attribute values.

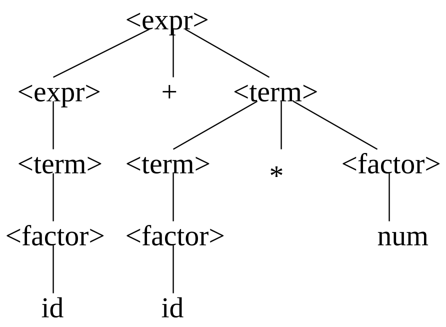
For example, consider the grammar:

<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → id | num | (<expr>)

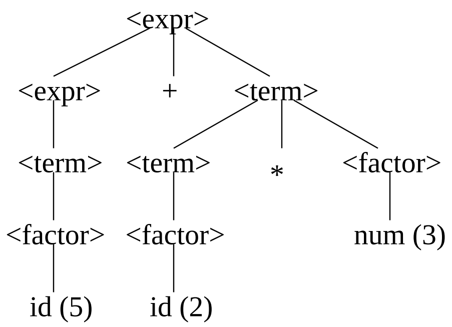
and expression:

x + y * 3

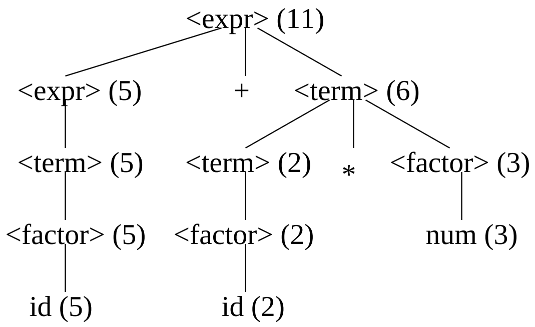
with parse tree:



Consider using an attribute named value to represent the mathematical result of an expression. If the value of x is 5 and y is 2, then the parse tree can be decorated/annotated with these intrinsic attribute values as follows:



Now synthesized attribute values can be computed for each of the nonterminals in the parse tree:



More practically, attributes can be used for type checking by adding a type attribute to each symbol. For example, consider the grammar:

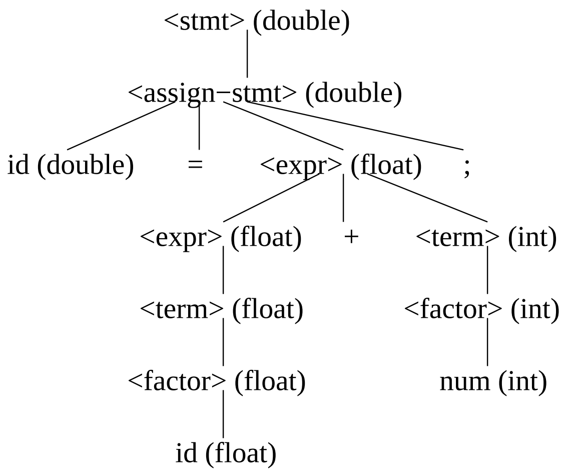
<stmt> → <assign-stmt>
<assign-stmt> → id = <expr> ;

<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → id | num | (<expr>)

and assignment statement:

x = y + 5;

and suppose that y is of type float and x is of type double. The annotated parse tree is:



Parser generators such as YACC and CUP generate parsers from attribute grammar specifications of the language being parsed.

3 Dynamic Semantics

Dynamic semantics refers to describing the behavior of programs as they execute. Several notations are used for dynamic semantics:

- operational semantics
- axiomatic semantics
- denotational semantics

3.1 Operational Semantics

Operational semantics:

- uses an interpreter for a language as a definition of the semantics of that language
- is usually done using an "idealized machine" or other notation that allows meaning to be defined without worrying about details of particular hardware etc.

3.2 Axiomatic Semantics

Axiomatic semantics uses logical pre- and postconditions to specify the meaning of programs (or parts of programs).

- a **precondition** is a condition that must be true for the program to execute correctly
- a **postcondition** is a condition that is guaranteed to be true after the program terminates, assuming that the precondition was satisfied

For example (factorial function):

```
int fact(int n) {
  // pre: n >= 0 /\ n < 13

 /* code for fact goes here */

  // post: result = n!
}
```

i.e. if function `fact` is called with a negative argument or an argument greater than 12, then nothing is guaranteed about the result. Otherwise, the function computes and returns the factorial.

Axiomatic semantics provides a set of axioms and inference rules for computing the most permissive (weakest) precondition necessary for a given program segment to satisfy a given postcondition. For example, for the program segment and postcondition:

```
 x = x + 1;
 // post: x > 3
```

the weakest precondition that ensures the postcondition holds is:

```
 // pre: x > 2
```

for integer `x`.

Note that `x > 5` is also a satisfactory precondition, but it is not the weakest precondition:

`x > 5` $\Rightarrow$ `x > 2`

`x > 2` $\not\Rightarrow$ `x > 5`

The axioms and inference rules can be used to prove that a program is correct - if the given precondition is stronger than (implies) the weakest precondition computed using the rules, then the program correctly implements the behavior specified by the pre- and postcondition.

Technically, this approach proves only **partial correctness** - if the program terminates (no infinite loops or recursion), then it is correct. A separate proof of termination is needed to show **total correctness**.

3.3 Denotational Semantics

Denotational semantics uses "semantic functions" that take programs as input and return the result of the program as output. This is similar to operational semantics, except that the notation used for defining the semantic functions is fixed and mathematically rigorous.

To give a denotational semantics for a language:

1. define the syntax of the language using BNF
2. define a representation for the results of executing "programs" in the language. This representation is often called *Answers*.
3. define a semantic function for each nonterminal in the BNF definition. Each such function has a case for each RHS of a BNF rule with the nonterminal on the LHS. These semantic functions define the meaning of each syntactic construct.

For example, consider the language of postfix (reverse Polish) expressions. Each binary arithmetic operator follows its operands.

Sample expressions and results:

`2 3 5 * +`, which is $2 + (3 \times 5)$ or 17 in normal arithmetic
`2 3 5 + *`, which is $2 \times (3 + 5)$ or 16 in normal arithmetic

For simplicity, we'll use numbers `1 - 5` and `+` and `*` only.

BNF definition:

<pe> → `1` | `2` | `3` | `4` | `5` | <pe> <pe> `+` | <pe> <pe> `*`

For this language, *Answers* is just the positive integers.

Since the BNF definition contains only one nonterminal and 7 rules, there is one semantic function $\mathrm{M_{pe}}$ with 7 cases.

Definition of $\mathrm{M_{pe}}$:

$\mathrm{M_{pe}}(\texttt{1}) = 1$
$\mathrm{M_{pe}}(\texttt{2}) = 2$
$\mathrm{M_{pe}}(\texttt{3}) = 3$
$\mathrm{M_{pe}}(\texttt{4}) = 4$
$\mathrm{M_{pe}}(\texttt{5}) = 5$
$\mathrm{M_{pe}}(\langle pe_0\rangle\ \langle pe_1\rangle\ \texttt{+}) = \mathrm{M_{pe}}(\langle pe_0\rangle) + \mathrm{M_{pe}}(\langle pe_1\rangle)$
$\mathrm{M_{pe}}(\langle pe_0\rangle\ \langle pe_1\rangle\ \texttt{*}) = \mathrm{M_{pe}}(\langle pe_0\rangle) \times \mathrm{M_{pe}}(\langle pe_1\rangle)$

Notes:

- typewriter font, i.e. `1` is used to distinguish syntax from mathematical entities such as 1
- when some nonterminal appears multiple times on the RHS of a rule, it is subscripted so that the occurrences can be distinguished

For "real" programming languages, the result of a program is usually contained in the values of various memory locations. A "snapshot" of these values at any point in the execution of the program is called a **state** of the program. State can be modeled as a function s from identifiers (variable names) to values. I.e. if the value of variable `foo` is 3, then $s(\texttt{foo}) = 3$.

To model state changes, we use **function override** notation: $[i \to v]s$ is a function that is exactly like s, except that it returns v when called with argument i. Formally:

$$[i \to v]s(i_2) = \text{if } i = i_2 \text{ then } v \\ \qquad\qquad\quad \text{else } s(i_2)$$

We use $\perp$ (bottom) to represent undefined or error. In the initial state, all identifiers map to $\perp$. Also, all of the semantic functions can return $\perp$ to indicate an error.

Consider the following BNF definition for a simple programming language:

<stmt> → id `=` <expr>; | `{`<sl>`}` | while `(`<expr>`)` <stmt>
<sl> → <stmt> <sl> | ϵ
<expr> → `0` | `1` | `2` | `3` | `4` | `5` | id | <expr> `+` <expr> | <expr> `<` <expr>

All values in this language are nonnegative integers (or whole numbers W), so s (the state) is a function from identifiers to $W \cup \{\perp\}$. $W_\perp$ is an abbreviation for $W \cup \{\perp\}$, and is pronounced "W lifted". Hence, the type of the state function s can be written as:

$$s : \text{identifiers} \to W_\perp$$

The semantic function $\mathrm{M_{stmt}}$ will return a state or $\perp$. For example, if a statement contains an undefined identifier, then the entire statement evaluates to $\perp$. Hence *Answers* = $\text{state}_\perp$, where
$\text{state} = \text{identifiers} \to W_\perp$.

Note that any set can be lifted.

The type signatures of the semantic functions are:

$\mathrm{M_{stmt}}$: <stmt> $\times\ \text{state}_\perp \to \text{state}_\perp$
$\mathrm{M_{sl}}$: <sl> $\times\ \text{state}_\perp \to \text{state}_\perp$
$\mathrm{M_{expr}}$: <expr> $\times\ \text{state}_\perp \to W_\perp$

Expressions only evaluate to whole numbers (or $\perp$) and can't cause state changes. However, $\mathrm{M_{expr}}$ must take a state as a parameter for looking up the values of any identifiers contained in the expression.

These functions are defined as follows:

```
$M_{\text{stmt}}(\text{id} = \text{<expr>};, s) =$
    if $s = \perp$ then $\perp$
    else if $M_{\text{expr}}(\text{<expr>}, s) = \perp$ then $\perp$
    else $[\text{id} \to M_{\text{expr}}(\text{<expr>}, s)]s$
$M_{\text{stmt}}(\{\text{<sl>}\}, s) = M_{\text{sl}}(\text{<sl>}, s)$
$M_{\text{stmt}}(\text{while (<expr>) <stmt>}, \text{s}) =$
    if $s = \perp$ then $\perp$
    else if $M_{\text{expr}}(\text{<expr>}, s) = \perp$ then $\perp$
    else if $M_{\text{expr}}(\text{<expr>}, s) = 0$ then $s$
    else $M_{\text{stmt}}(\text{while (<expr>) <stmt>}, M_{\text{stmt}}(\text{<stmt>}, s))$

$M_{\text{sl}}(\text{<stmt> <sl>}, s) =$
    if $s = \perp$ then $\perp$
    else $M_{\text{sl}}(\text{<sl>}, M_{\text{stmt}}(\text{<stmt>}, s))$
$M_{\text{sl}}(\epsilon, s) = s$

$M_{\text{expr}}(0, s) = 0$
$M_{\text{expr}}(1, s) = 1$
$M_{\text{expr}}(2, s) = 2$
$M_{\text{expr}}(3, s) = 3$
$M_{\text{expr}}(4, s) = 4$
$M_{\text{expr}}(5, s) = 5$
$M_{\text{expr}}(\text{id}, s) =$
    if $s = \perp$ then $\perp$
    else $s(\text{id})$
$M_{\text{expr}}(\text{<expr}_0\text{>} + \text{<expr}_1\text{>}, s) =$
    if $s = \perp$ then $\perp$
    else if $M_{\text{expr}}(\text{<expr}_0\text{>}, s) = \perp$ or $M_{\text{expr}}(\text{<expr}_1\text{>}, s)$ then $\perp$
    else $M_{\text{expr}}(\text{<expr}_0\text{>}, s) + M_{\text{expr}}(\text{<expr}_1\text{>}, s)$
$M_{\text{expr}}(\text{<expr}_0\text{>} < \text{<expr}_1\text{>}, s) =$
    if $s = \perp$ then $\perp$
    else if $M_{\text{expr}}(\text{<expr}_0\text{>}, s) = \perp$ or $M_{\text{expr}}(\text{<expr}_1\text{>}, s)$ then $\perp$
    else if $M_{\text{expr}}(\text{<expr}_0\text{>}, s) < M_{\text{expr}}(\text{<expr}_1\text{>}, s)$ then 1
    else 0
```

Note that iteration is defined recursively, and may not terminate (for an infinite loop).

Denotational semantics allows the use of let to define a "local variable" so that the same expression need not be written out multiple times. For example:

```
$M_{\text{stmt}}(\text{id} = \text{<expr>};, s) =$
    if $s = \perp$ then $\perp$
    else let $v = M_{\text{expr}}(\text{<expr>}, s)$ in
        if $v = \perp$ then $\perp$
        else $[\text{id} \to v]s$
```

Uses of denotational semantics:

- formal definition of language semantics
- proving properties of languages
- automated generation of compilers/interpreters