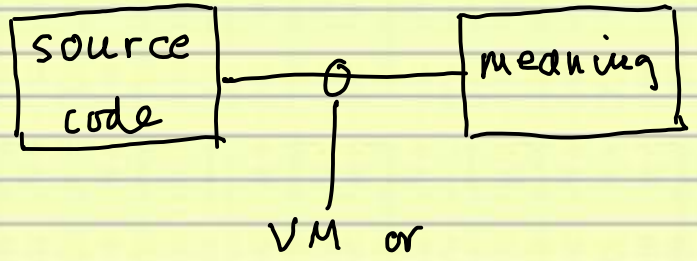
Semantics



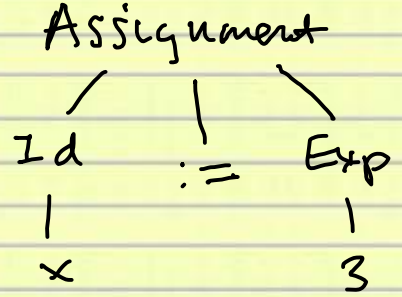
Precursor to semantics is SYNTAX.

Syntax can be concrete or abstract

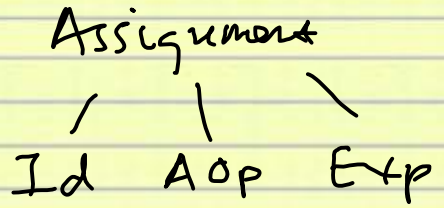
e.g. assignment in C is $x = 3$

" " Pascal is $x := 3$

semantically these are same, so it would be nice to use syntactic definition that helps the semantics.



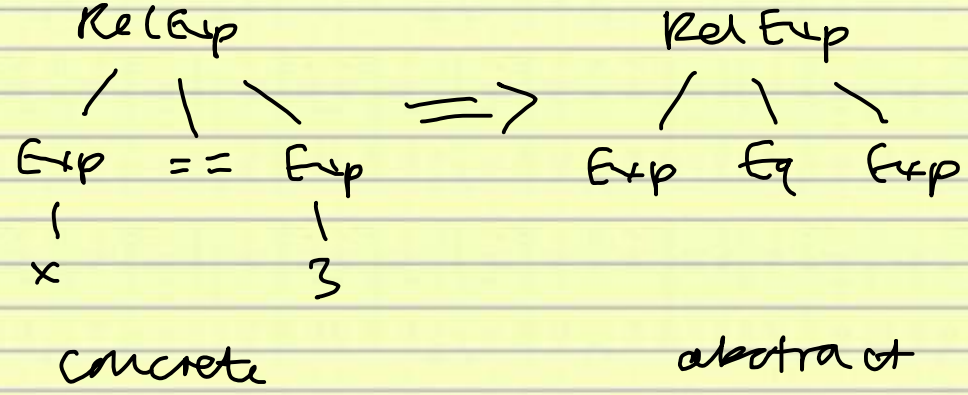
Abstract syntax replaces concrete symbols with categories or tokens.



← abstract, replacing concrete symbols with tokens

In abstract syntax, every leaf is a token, there are no concrete symbols, although very often for readability, we put them back in.

Another e.g. `x == 3` in C
`x = 3` in Pascal



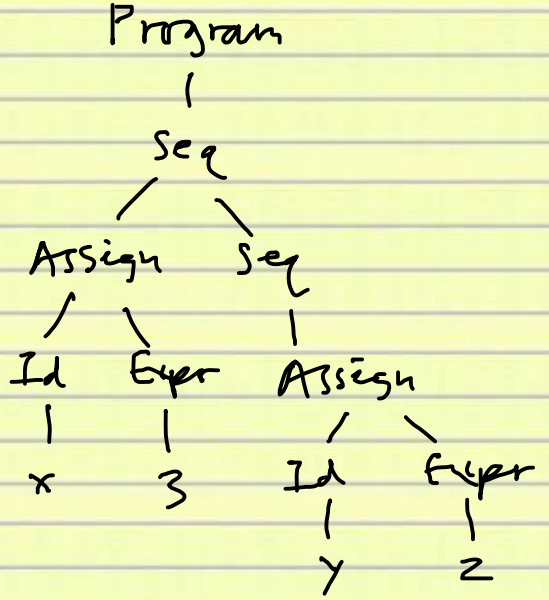
could even drop the Eq operator if we label nodes correctly

We will use JavaCC (compiler-compiler) which is a parser generator. Given an abstract syntax definition we will generate a parser for that language.

We will then add code to interpret the program to produce results.

The parser takes a source code program as input, and generates Abstract Syntax Trees (ASTs)

e.g



this might come from

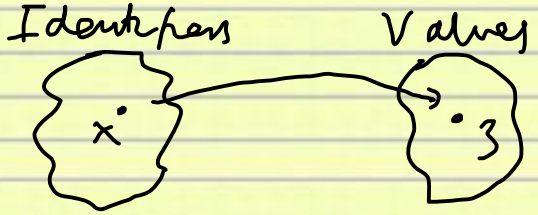
$x = 3;$

$y = 2$

Denotational semantics

e.g. what does $x = 3$ mean?

answer is it changes the value bound to x



Each syntactic constructs is going to map to either a function or a set, mostly functions.

If the language of functions and sets is understood, then we therefore understand the programming language.

The functions are expressed in the lambda calculus.

Lambda calculus.

Functions are written as $\lambda x . B$

parameter, any expression

We could have named the function and written $f(x) = B$

name

The λ form is anonymous

Constants are written as $a, b, c, \ldots$

Parameter are written as $x, y, z \ldots$

An application is written as $E_1\ E_2$ where these are both expression

Expressions are constants, parameters, functions or applications. Add parentheses as necessary.

e.g. $(\lambda x.x)\ a$

This applies the function $\lambda x.x$ to the constant a

We bind the parameter to its argument (expression after the function) and then substitute the binding for all occurrences of the parameter in the body of the function.

e.g. $(\lambda x.x)\ a$

$= x[x \backslash a]$

$= a$

e.g. $(\lambda x.x\ b)(\lambda x.x)$

$= (x\ b)[x \backslash (\lambda x.x)]$

$= (\lambda x.x)\ b$

$= x[x \backslash b] = b$

This is β-reduction and it reduces an expression to its simplest

Problem when a parameter is used twice in a nested expression.

$$(\lambda x.(\lambda x.x))\ a$$

$$(\lambda x.x)[x \backslash a] = (\lambda x.x)$$

Do not replace x if it is <u>bound</u> in a function, only if it is free

Sometimes an application cannot be reduced;

$$(\lambda x.\ x\ a)\ b$$

$$= (x\ a)[x \backslash b]$$

$$= b\ a$$

Sometimes we use arithmetic (not pure Lambda calculus)

$$(\lambda x.(\lambda y.x+y))\ 2\ 3 = (\lambda y.2+y)\ 3 = 2+3 = 5$$

The language we will look at is very simple:

The abstract syntax is:

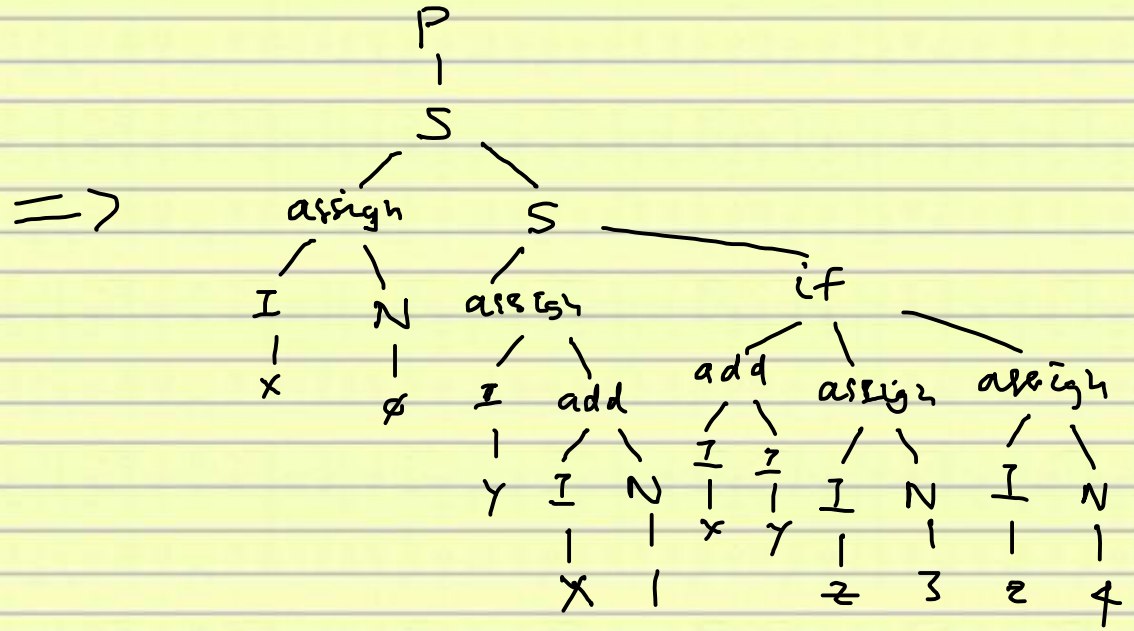
$$P ::= S$$

$$S ::= S_1 ; S_2 \mid I = E \mid$$
if E then S_1 else S_2 |
while E do S

$$E ::= I \mid N \mid E_1 + E_2 \ldots$$

Sample program

```
x = 0;
y = x+1;
if x+y then
  z = 3
else
  z = 4
```



The semantics are a set of equations, one per syntactic construct, that maps syntax to a function.
Syntax is put in $[\![\;]\!]$

e.g. $E[\![I]\!] = \lambda s.\, s([\![I]\!])$

s is store which maps identifiers to numbers

e.g. $s_0 = \{(x, \emptyset)\}$

$$E[\![x]\!]\, s_0 = (\lambda s.\, s[\![x]\!])\, s_0$$
$$= s_0[\![x]\!]$$
$$= \emptyset$$