More on CFGs

In this class, we're mostly interested in using CFGs to define the "tree structure" of program syntax (ASTs).

A CFG is a list of equations of the form

```
<variable> ::= <RHS>
```

- The LHS variable is called non-terminal, represents a tree type
- The RHS specifies one way to build such a tree (a node type)

Example

```
expr ::= <RHS>
```

Means there's an expr tree type, built according to RHS

RHS = sequence of variables or "terminals"

- A RHS variable x means there's should be a subtree of type x
- A terminal either carries primitive data, or suggests a piece of concrete syntax

```
expr ::= NUM
```

- NUM is a terminal symbol with integer primitive data
- So the node type carries an int.

```
expr ::=
    expr + expr
```

- The two expr variables mean the "+ node" must have two subtrees of type expr
- The + suggests that this node should represents an addition in the concrete syntax

If there multiple equations (aka production rules) with the same LHS variable, we merge the RHS using "|" (vertical bars) for brevity.

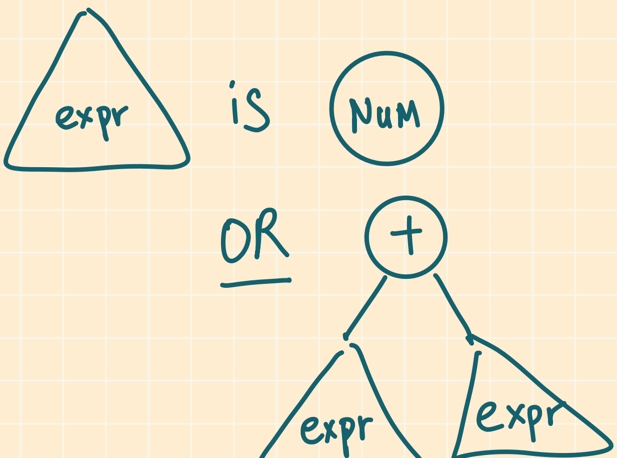
```
expr ::= NUM

expr ::=
    expr + expr
```

$\Rightarrow$

```
expr ::= NUM
       | expr + expr
```

~~| (expr)~~



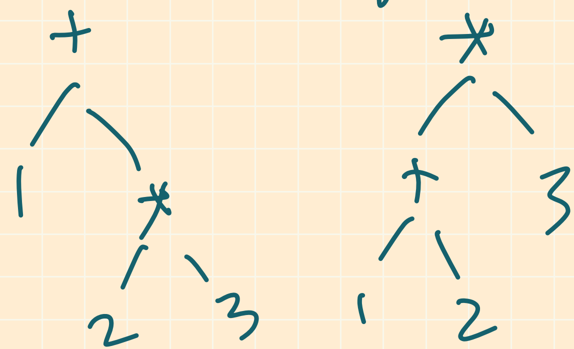
So this CFG says:

- We have expr trees in our language.
- An expr tree can be built from a NUM node (carrying an int value).
- An expr tree can also be built from a "+ node" (carrying two expr subtrees)

FAQs

- Does the order of production rules matter? No. Every rule is an equally valid alternative.
- Should have a rule for parentheses? No, trees are already unambiguous.

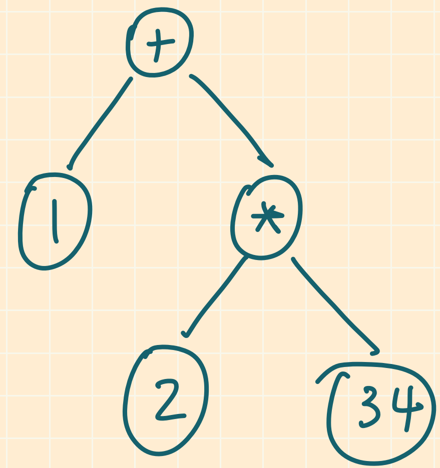
$(1+2) * 3$



Parser turns a string into an AST according to a CFG.

- parsing algorithms can be super complex
- CS 160 spends 5 weeks just on parsing algorithms !!!
- In CS 162, you only need to **manually parse** simple programs into ASTs.

"1 + 2 * 34" →



Manual parsing

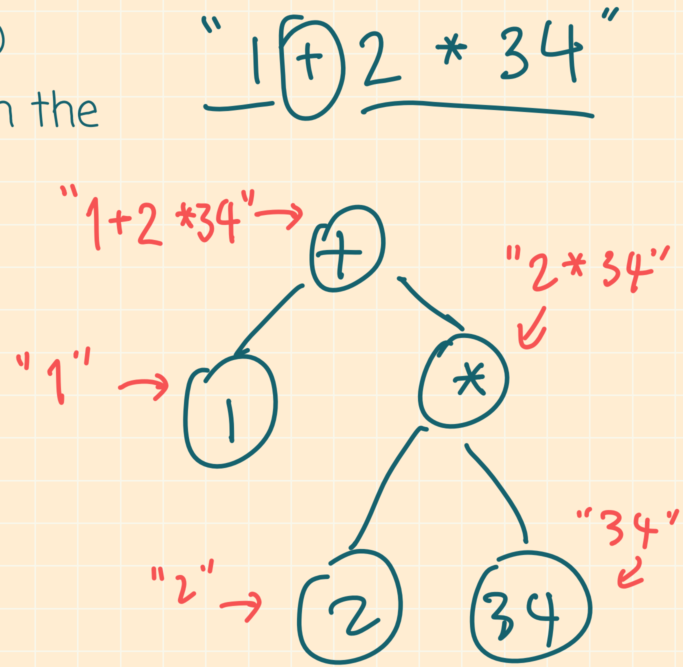
Intuitions:

1. Look at the "outermost" structure a program (an operator with the lowest precedence)
2. Find a CFG rule whose RHS matches with the "outermost" structure
3. Create an AST node using that rule
4. Go to step 1 and recurse.

```
expr ::= NUM
       | expr + expr
       | expr * expr
```

Assume + has lower precedence than *

Example



Exercise:

Manually parse the proposition "$P \to (Q \land R \lor S)$" into an AST according to the abstract syntax:

```
prop ::= VAR
       | prop → prop
       | prop ∧ prop
       | prop ∨ prop
```

Assuming that $\land > \lor > \to$ for precedence

Summary

	CS 138 Concrete syntax	CS 162 Abstract syntax
Used by	Programmers to write code	Language designers to analyze programs
Described by	CFG	CFG
Programs represented by	Strings	ASTs

CS 160

How to represent ASTs using classes and objects

(We'll use Python's @dataclass exclusively in CS 162)

Recipe:

- every LHS non-terminal becomes an empty superclass
- every rule becomes a subclass
- every RHS non-terminal becomes a child field
- leaf nodes may contain primitive data

```python
@dataclass
class Expr:
  pass
```

```
expr ::= NUM
       | expr + expr
       | expr - expr
       | expr * expr
```

```python
@dataclass
class Add(Expr):
  e1: Expr
  e2: Expr

@dataclass
class Num(Expr):
  value: int
```

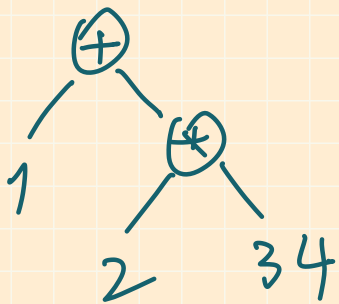
How to represent ASTs in Python

Exercise: represent the ASTs for Boolean propositional formulas in Python.

How to construct ASTs in Python

Just call the class constructors!

Example: write down a Python expression that represents the AST for "1+2*34"



Python object:

```
Add( Num(1),
     Mul( Num(2),
          Num(34))
)
```

Exercise: write down the Python expression that represents the AST for "P → false ∧ (Q ∨ true)"

How to traverse ASTs represented as @dataclass

We'll exclusively use pattern matching + recursion in CS 162. (No visitors)

Example: is this Expr an add node?

Example: count the number of AST nodes

Exercise: compute the height of a Prop AST

Semantics

The meaning of a program, or
"What happens if I click the run button"

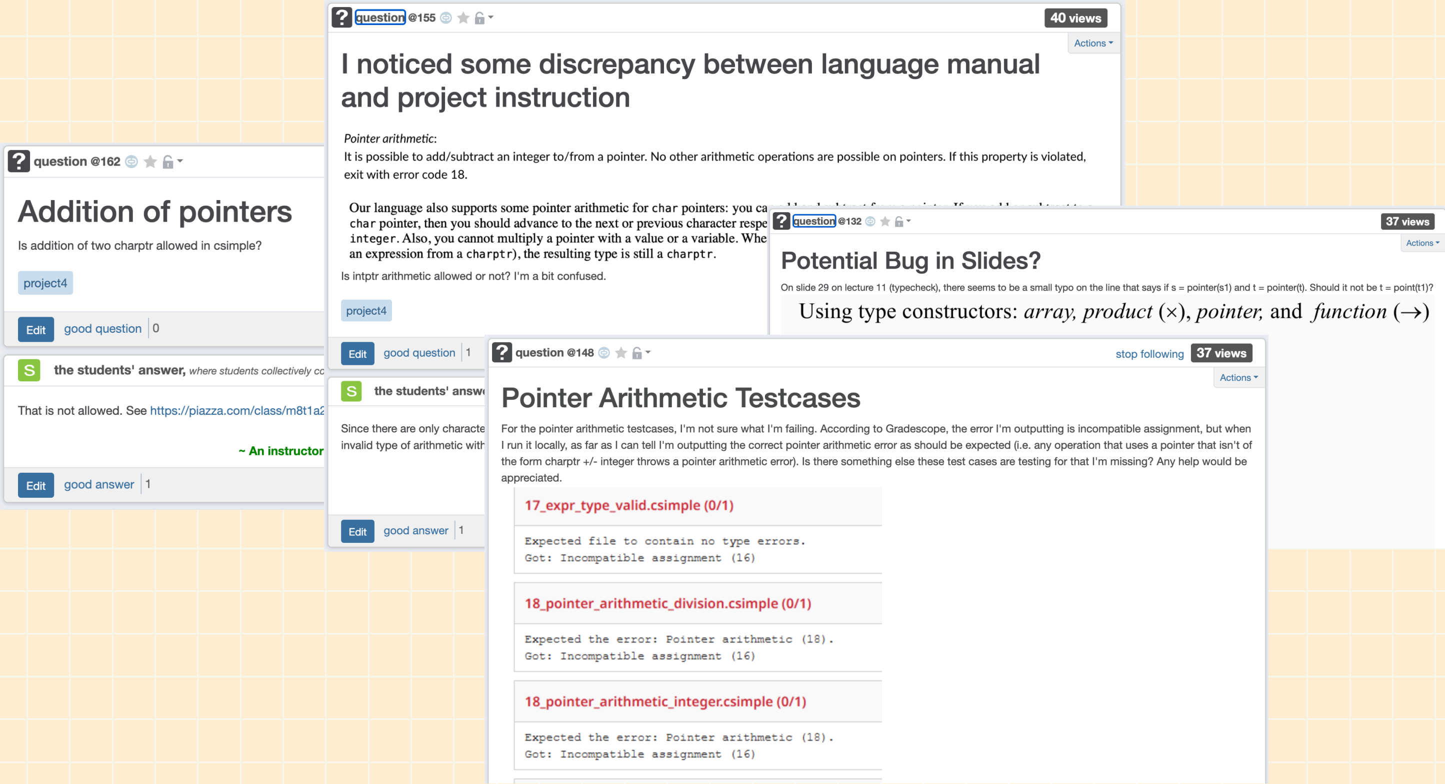
Semantics specified through natural language

CS 160's language specification:

Our language also supports some pointer arithmetic for char pointers: you can add and subtract from a pointer. If you add or subtract to a char pointer, then you should advance to the next or previous character respectively.

Can you think of any ambiguities or edge cases that are not specified by the text?

I was a TA...


question @162
Addition of pointers
Is addition of two charptr allowed in csimple?
project4
Edit
good question
0
the students' answer, where students collectively co
That is not allowed. See https://piazza.com/class/m8t1a2
~ An instructor
Edit
good answer
1
question @155
40 views
Actions
I noticed some discrepancy between language manual and project instruction
Pointer arithmetic:
It is possible to add/subtract an integer to/from a pointer. No other arithmetic operations are possible on pointers. If this property is violated, exit with error code 18.
Our language also supports some pointer arithmetic for char pointers: you ca
char pointer, then you should advance to the next or previous character respe
integer. Also, you cannot multiply a pointer with a value or a variable. Whe
an expression from a charptr), the resulting type is still a charptr.
Is intptr arithmetic allowed or not? I'm a bit confused.
project4
Edit
good question
1
the students' answ
Since there are only characte
invalid type of arithmetic with
Edit
good answer
1
question @132
37 views
Actions
Potential Bug in Slides?
On slide 29 on lecture 11 (typecheck), there seems to be a small typo on the line that says if s = pointer(s1) and t = pointer(t). Should it not be t = point(t1)?
Using type constructors: array, product (×), pointer, and function (→)
question @148
stop following
37 views
Actions
Pointer Arithmetic Testcases
For the pointer arithmetic testcases, I'm not sure what I'm failing. According to Gradescope, the error I'm outputting is incompatible assignment, but when I run it locally, as far as I can tell I'm outputting the correct pointer arithmetic error as should be expected (i.e. any operation that uses a pointer that isn't of the form charptr +/- integer throws a pointer arithmetic error). Is there something else these test cases are testing for that I'm missing? Any help would be appreciated.
17_expr_type_valid.csimple (0/1)
Expected file to contain no type errors.
Got: Incompatible assignment (16)
18_pointer_arithmetic_division.csimple (0/1)
Expected the error: Pointer arithmetic (18).
Got: Incompatible assignment (16)
18_pointer_arithmetic_integer.csimple (0/1)
Expected the error: Pointer arithmetic (18).
Got: Incompatible assignment (16)

Semantics specified through implementation

CS 160's language supports procedures defined in sequence:

```
procedure foo() { … }
procedure bar() { … }
…
```

Here's the C++ code that unambiguously defines what type of procedures is valid.

Q: Are mutual recursion supported?

Combine natural language + implementation?

Our language also supports some pointer arithmetic for char pointers: you can add and subtract from a pointer. If you add or subtract to a char pointer, then you should advance to the next or previous character respectively.

in sync

```
void add_proc_symbol(ProcImpl *p)
    {
        char *name = strdup(p->m_symname->spelling());
        Symbol *s = new Symbol(), *exists = m_st->lookup(name);
        s->m_basetype = bt_procedure;

        if (exists != NULL && exists->get_scope() == m_st->get_scope())
        {
            this->t_error(dup_proc_name, p->m_attribute);
        }

        if (!strcmp(name, "Main") && p->m_decl_list->size() > 0)
        {
            this->t_error(nonvoid_main, p->m_attribute);
        }

        m_st->open_scope();
        p->m_attribute.m_scope = m_st->get_scope();

        // Read in the argument list
        for (std::list<Decl_ptr>::iterator decl_iter = p->m_decl_list->begin();
             decl_iter != p->m_decl_list->end(); decl_iter++)
        {
            DeclImpl *dip = dynamic_cast<DeclImpl *>(*decl_iter);
            dip->accept(this);

            // Add the types for each variable of that type
            Basetype bt = dip->m_type->m_attribute.m_basetype;
            for (std::list<SymName_ptr>::iterator sym_iter = dip->m_symname_list->begin();
                 sym_iter != dip->m_symname_list->end(); sym_iter++)
                s->m_arg_type.push_back(bt);
        }

        // Read in return type
        p->m_type->accept(this);
        s->m_return_type = p->m_type->m_attribute.m_basetype;

        // Make the procedure known
        m_st->insert_in_parent_scope(name, s);

        // Check the procedure body
        p->m_procedure_block->accept(this);

        m_st->close_scope();
    }
```

Ways of Specifying Semantics

	Unambiguous	Readable	Maintainable
Natural language	✗	✓	✓
Implementation	✓	✗	✓
Natural language + implementation	✓	✓	✗

Inference Rules are the secret language of PL designers

A bunch of stuff

Some stuff

T-StrTpl
$$\frac{\Gamma, \mathsf{tpl}\ \mathsf{String} \vdash t : \mathsf{String}\ \mathsf{list}}{\Gamma \vdash \mathbf{strtpl}\ t : \mathsf{String}}$$

T-TreeTpl
$$\frac{\Gamma, \mathsf{tpl}\ \mathsf{NodeTy} \vdash t : \mathsf{NodeTy}\ \mathsf{list}}{\Gamma \vdash \mathbf{treetpl}\ t : \mathsf{NodeTy}\ \mathsf{list}}$$

T-NilTpl
$$\frac{}{\Gamma, \mathsf{tpl}\ \tau \vdash [\,] : \tau\ \mathsf{list}}$$

T-ConsTpl
$$\frac{\Gamma, \mathsf{tpl}\ \tau \vdash p : \tau \qquad \Gamma, \mathsf{tpl}\ \tau \vdash ps : \tau\ \mathsf{list}}{\Gamma, \mathsf{tpl}\ \tau \vdash (p :: ps) : \tau\ \mathsf{list}}$$

TP-Str
$$\frac{}{\Gamma, \mathsf{tpl}\ \tau \vdash s : \tau}$$

TP-Node
$$\frac{\Gamma, \mathsf{tpl}\ \tau \vdash t : \tau\ \mathsf{list} \qquad \forall i.\ \Gamma \vdash e_i : \mathsf{String}}{\Gamma, \mathsf{tpl}\ \tau \vdash \mathbf{node}\ (s, [(s_i, e_i)*], t) : \tau}$$

wbHoare-create-stack
$$\frac{\forall N.\ \mathit{stack}_{\bullet}(N, [\,]) * \mathit{stack}_{\circ}(N, [\,]) \vdash {\Rrightarrow} R(N) * \exists s.\ \mathit{stack}_{\bullet}(N, s) \qquad \forall N.\ (\!|P * R(N)|\!)\ e\ (\!|\Phi|\!)^{\mathcal{O}}}{(\!|P|\!)\ e\ (\!|\Phi|\!)^{\mathcal{O}}}$$

wbHoare-access-stack
$$\frac{N \notin \mathcal{O}}{\left(\forall s.\ (\!|P * \mathit{stack}_{\bullet}(N, s)|\!)\ e\ (\!|x.\ \Phi(x) * \mathit{stack}_{\bullet}(N, s)|\!)^{\mathcal{O} \cup \{N\}}\right) \vdash (\!|\mathit{stack}_{\exists}(N) * P|\!)\ e\ (\!|\Phi|\!)^{\mathcal{O}}}$$

wbHoare-mend-stack
$$(\!|P|\!)\ e\ (\!|\Phi|\!)^{\mathcal{O} \setminus \{N\}} \vdash (\!|\mathit{stack}_{\bullet}(N, s) * P|\!)\ e\ (\!|x.\ \Phi(x) * \mathit{stack}_{\bullet}(N, s)|\!)^{\mathcal{O}}$$

Hoare-wbHoare
$$\{P\}\ e\ \{\Phi\} \vdash (\!|P|\!)\ e\ (\!|\Phi|\!)^{\mathcal{O}}$$

2.1.4 *Lambda Abstractions.* Unlike numbers and variables, there are explicit synthesis and analysis rules for unmarked lambda abstractions. This is because expected input and output types are known, and we may verify that the type annotation and body match them.

USLam
$$\frac{\Gamma,\ x : \tau_1 \vdash_U e \Rightarrow \tau_2}{\Gamma \vdash_U \lambda x : \tau_1.\ e \Rightarrow \tau_1 \to \tau_2}$$

UALam
$$\frac{\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \tau \sim \tau_1 \qquad \Gamma,\ x : \tau \vdash_U e \Leftarrow \tau_2}{\Gamma \vdash_U \lambda x : \tau.\ e \Leftarrow \tau_3}$$

By the end of this lecture, you'll know how to speak this secret tongue.

Inference Rules are great

Precise: formal semantics - "mathy"

Readable: concise, easy to read

Maintainable: hmm

Adoption: Who uses this sh*t besides PL researchers?

Type checking rules for lambda calculus (1930s)

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \text{tvar}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}\ \to\text{-intro} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \to\text{-elim}$$

WebAssembly (aka Wasm, 2017-)

Native-speed assembly language on web browsers

Supported in every major browser

"Hmm, I know formal semantics are great, but few people in industry know it.

Let's get it right this time."

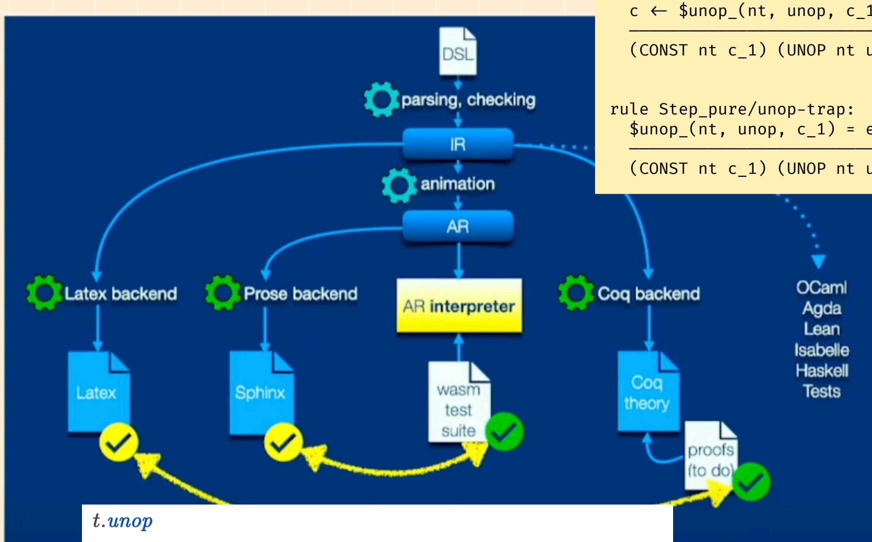
Andreas Rossberg

(Researcher working on Wasm)

WebAssembly (aka Wasm, 2017-)

Native-speed assembly language on web browsers

Supported in every major browser




```
rule Step_pure/unop-val:
  c ← $unop_(nt, unop, c_1)
  ────────────────────────────────────────────────
  (CONST nt c_1) (UNOP nt unop)  ↝  (CONST nt c)


rule Step_pure/unop-trap:
  $unop_(nt, unop, c_1) = eps
  ────────────────────────────────────────────────
  (CONST nt c_1) (UNOP nt unop)  ↝  TRAP
```

https://github.com/Wasm-DSL/spectec/blob/main/specification/wasm-3.0/4.3-execution.instructions.spectec

Formal spec

Machine-checked math proof

```
Theorem t_preservation: forall s f es s' f' es' ts hs hs',
    reduce hs s f es hs' s' f' es' →
    config_typing s (f, es) ts →
    config_typing s' (f', es') ts.
Proof.
  …
QED.
```

https://github.com/WasmCert/WasmCert-Coq/blob/master/theories/type_preservation.v

Coq/Rocq

Prose

t.*unop*

1. Assert: due to validation, a value of value type t is on the top of the stack.
2. Pop the value t. const c_1 from the stack.
3. If $unop_t(c_1)$ is defined, then:
   a. Let c be a possible result of computing $unop_t(c_1)$.
   b. Push the value t. const c to the stack.
4. Else:
   a. Trap.

https://github.com/Wasm-DSL/spectec/blob/main/specification/wasm-3.0/4.3-execution.instructions.spectec

Relation = a Google Sheet table (no duplicate rows)

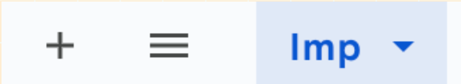
Name = table name

Membership: An element (x, y, ...) is in the relation if the table has a row (x, y, ...)

Arity = the number of columns

Size = the number of rows

P	Q	R
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE



$(\mathrm{TRUE}, \mathrm{FALSE}, \mathrm{TRUE}) \notin \mathrm{Imp}$

" $\Rightarrow$ "

" $0+1 \Rightarrow 1$ " means

$(0+1, 1) \in \boxed{\Rightarrow}$

Program	Value
0+1	1
1*2	2
1+2*3	7

$(0+1, 2) \notin$ " $\Rightarrow$ "

If one column contains all programs, and the other column contains their final results, we've fully specified the semantics of the language

If you want to know how a program behaves, look up its value in the table

Inference rules - A finite way to define infinite-size relations

We call a statement about relation membership a judgment.

minecraft resource (MC)

A rule has the form: MC recipe

- a horizontal bar
- a rule name on the side
- premises: a (possibly-empty) sequence of judgments above the bar
- conclusion: a single judgment below the bar

required resource

RecipeName

$$\frac{\text{Judgment1} \quad \text{Judgment2} \quad \text{Judgment3} \ldots}{\text{Judgment4}}\ \text{RuleName}$$

produced resource

If you can show all premises hold, the rule allows you prove the conclusion

An **axiom** is a rule with zero premise → recipe that makes a resource out of thin air

A **relation** is defined by a **set** of rules.

Rules:

$$\frac{}{\text{coal}}\ \text{Mine} \qquad \frac{}{\text{wood}}\ \text{Chop}$$

$$\frac{\text{coal} \quad \text{wood}}{\text{torch}}\ \text{MakeTorch}$$

To prove that (x, y, ...) in relation R:

- we start from the judgment R(x, y, ...)
- **derivation**: we keep applying applicable rules bottom-up to obtain new premises
- the judgment is proved if no more premises need to be proved (always end with axioms)

The resulting trace of how we applied rules is called a **derivation tree**

Derivation tree showing you can make a torch:

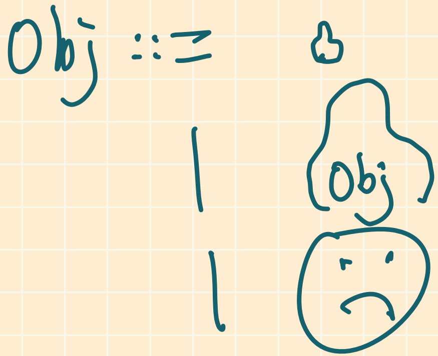
$$\boxed{\dfrac{\dfrac{}{\text{coal}}\,\text{Mine} \quad \dfrac{}{\text{wood}}\,\text{Chop}}{\text{torch}}\,\text{Make Torch}}$$

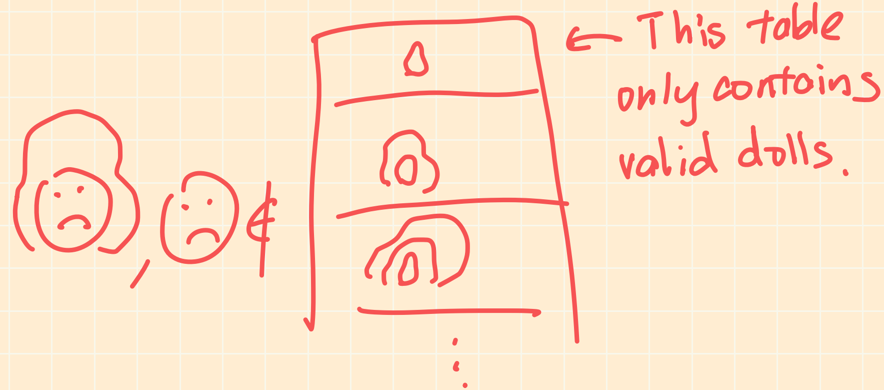
Example 1: Russian nesting dolls

Relation/judgment: is <object> a Russian doll?

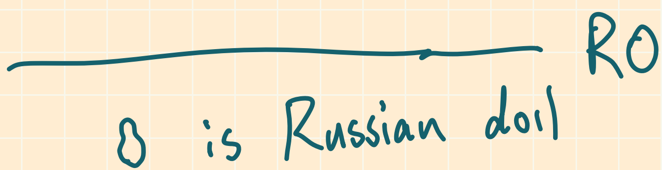
arity = 1

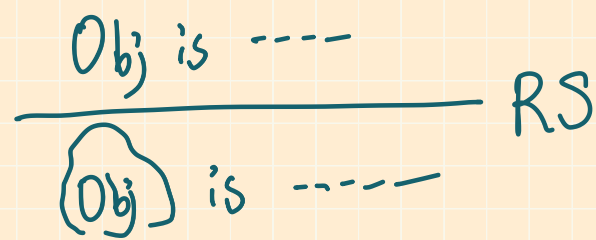
<object> is given by the CFG:





Rules:





Rules:

$$\dfrac{}{\quad \text{is Russian doll}} \; RO \qquad \dfrac{Obj \text{ is ----}}{Obj \text{ is ----}} \; RS$$

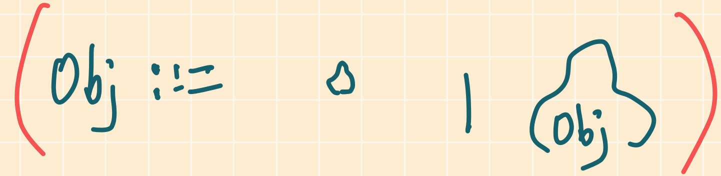
Derivation tree:

$$\dfrac{\dfrac{\dfrac{}{\quad \text{is ----}}\; RO}{\quad \text{is ----}}\; RS}{\quad \text{is Russian doll}}\; RS$$

Label each application of a rule w/ its name

Example 2: more Russian dolls

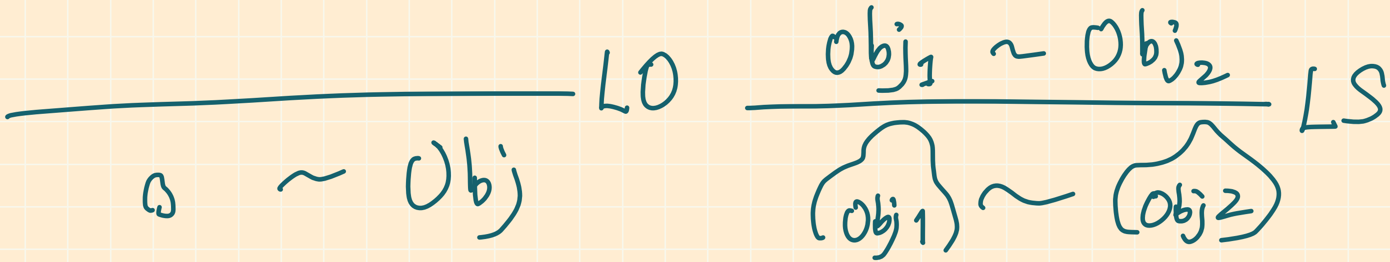
Mystery relation/judgment:



$$\text{Obj} \sim \text{Obj}$$

arity = 2

Rules:



Let's prove: