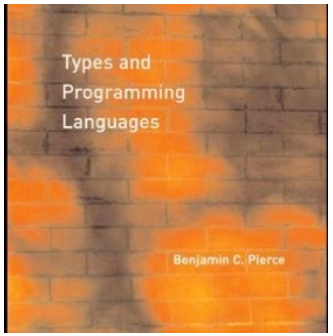
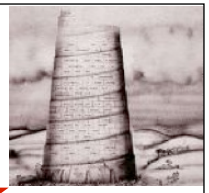
Preliminaries

CS 565

Lecture 2



Basics



- Ordered sets:
  - A binary relation R on a set S is:

    reflexive if $(s,s) \in R$

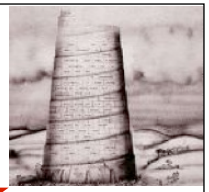
    symmetric if $(s,t) \in R \wedge (t,s) \in R$

    transitive if $(s,t), (t,u) \in R \Rightarrow (s,u) \in R$

    antisymmetric if $(s,t), (t,s) \in R \wedge s = t$
  - A reflexive, transitive relation on S is a called a *preorder* on S
  - A reflexive, transitive, antisymmetric relation on S is called a *partial order* ($\sqsubseteq$) on S

    A partial order is a *total order* if for each $s,t \in S$, either $s \sqsubseteq t$ or $t \sqsubseteq s$.
  - A reflexive, transitive, symmetric relation on S is called an *equivalence relation* on S.

Basics



- Suppose R is a binary relation on S.
  - The *reflexive closure* of R is the smallest reflexive relation R' that contains R.
  - The *transitive closure* of R is the smallest transitive relation R' that contains R.
- Let R be a binary relation on S. Define R' as:

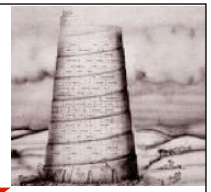
  $R' = R \cup \{(s,s) \mid s \in S\}$

  Show that R' is the reflexive closure of R.

  Need to show that R' is a reflexive relation on R.

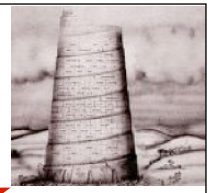
  Need to show that R' is the smallest such relation.

Basics



- Let S have preorder $\sqsubseteq$. We say $\sqsubseteq$ is *well-founded* if it contains no infinite decreasing chains.
- A preorder is a relation that is reflexive and transitive.
  - The preorder defining the natural numbers is well-founded.
  - The preorder on the integers is not.

Induction

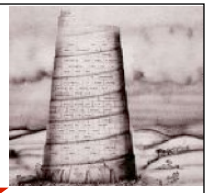


- Principle of ordinary induction on natural numbers:
  - Suppose that P is a predicate on N.
  - Then if P(0) holds, and for all i, P(i) ⇒ P(i+1),

    P(n) holds for all n.
- Example:

  Theorem: $2^0 + 2^1 + \cdots + 2^{n-1} = 2^{n+1} - 1 \text{ for all } n$

Proof



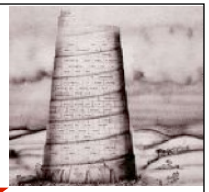
- By induction on n.
  - Base case (n = 0):

$$2^0 = 2^1 - 1$$

  - Inductive case (n = i + 1):

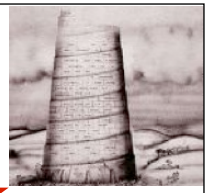
$$\begin{aligned} 2^0 + 2^1 + \cdots + 2^{i+1} &= \\ (2^0 + 2^1 + \cdots + 2^i) + 2^{i+1} &= \\ 2^{i+1} - 1 + 2^{i+1} &= \quad \text{(induction hypothesis)} \\ 2 * 2^{i+1} - 1 &= \quad 2^{i+2} - 1 \end{aligned}$$

Goals



- Introduce a simple well-known language
  - basic arithmetic expressions
- Study properties of this language via
  - abstract syntax
  - inductive definitions
  - proof strategies
- Focus on techniques to reason about a language rather than the language itself

Syntax



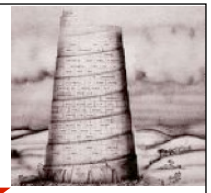
- BNF Grammar:

`t ::=`	terms
`true`	constant true
`| false`	constant false
`| if t then t else t`	Conditional
`| 0`	constant 0
`| succ t`	successor
`| pred t`	predecessor
`| iszero t`	zero test

- Terminology:

  `t` is a *metavariable*, not a variable of the object language

Programs



A program is just a term built from the grammar:

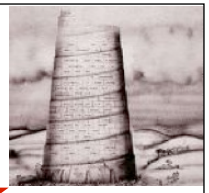
```
true → true
if false then 0 else succ 0 → 1
iszero (pred (succ 0)) → true
succ(succ(succ 0))) → 3
```

Grammar does not prevent writing terms that may not make much sense:

```
succ true → ?
if 0 then 0 else 0 → ?
```

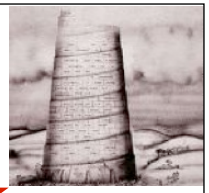
Grammar does not define rules to guide us in ascribing translation or meaning to terms

Abstract vs. concrete syntax



- What does the grammar actually define?
  1. a set of character strings
  2. a set of tokens
  3. a set of abstract syntax trees
- It defines all three, but we are most interested in (3)
  - Call the grammar an “abstract grammar” because it defines a set of abstract syntax trees, along with a strategy for mapping character strings to these trees.
  - We use parentheses to disambiguate terms when the intended corresponding tree is not clear from context

Abstract vs. concrete syntax



Are:

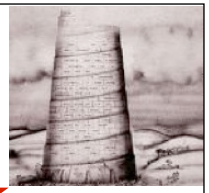
```
succ 0
succ (0)
((succ ((((0))))))
```

“the same term”?

What about:

```
succ 0
pred (succ (succ 0))
```

Syntax



- The grammar is shorthand for the following inductive definition:
- Definition: The set of terms is the smallest T such that

  {`true,false,0`} ⊆ T

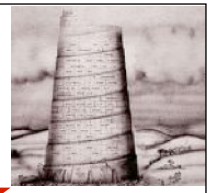
  if `t1` ∈ T then {`succ t1,pred t1,iszero t1`} ⊆ T

  if `t1`, `t2` ∈ T, and `t3` ∈ T, then

  {if `t1` then `t2` else `t3`} ⊆ T

- First clause says there are three simple expressions (i.e., expressions that do not refer to meta-variables) in T
- Second and third clauses says how compound expressions can be constructed from smaller constituent pieces

Alternative formulation: Inference Rules



$$\texttt{true} \in \mathsf{T} \qquad \texttt{false} \in \mathsf{T} \qquad \texttt{0} \in \mathsf{T}$$

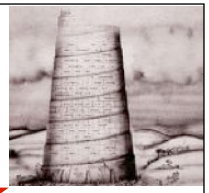
$$\frac{t \in \mathsf{T}}{\texttt{succ } t \in \mathsf{T}} \qquad \frac{t \in \mathsf{T}}{\texttt{pred } t \in \mathsf{T}}$$

$$\frac{t \in \mathsf{T}}{\texttt{iszero } t \in \mathsf{T}} \qquad \frac{t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, t_3 \in \mathsf{T}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \in \mathsf{T}}$$

These rules are often referred to as “inference” rules
Rules without premises are called axioms

Each rule is read “If we have established the statements in the premises listed above the line, then we may conclude the statement listed below the line.”

Alternative formulation



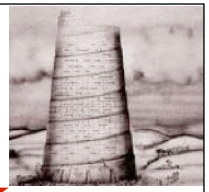
- For each natural number i, define set S_i as follows:

$$
\begin{aligned}
S_0 &= \emptyset \\
S_{i+1} &= \{\texttt{true}, \texttt{false}, \texttt{0}\} \cup \\
&\quad \{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1 \,|\, \texttt{t}_1 \in S_i\} \cup \\
&\quad \{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \,|\, \texttt{t}_1, \texttt{t}_2, \texttt{t}_3, \in S_i\} \\
S &= \bigcup_i S_i
\end{aligned}
$$

- This definition is constructive – it gives an explicit procedure for generating all the elements of T

  Exercise: How many elements does S_3 have? What about S_i for arbitrary i? Show that for any i, $S_i \subseteq S_{i+1}$.

Equivalence



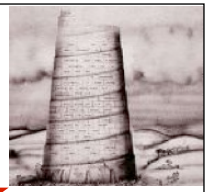
- We have seen two basic ways for describing the language of simple arithmetic:
  - inductively where T is the smallest set closed under certain rules

    BNF shorthand

    explicit inductive definition

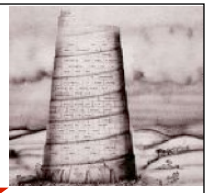
    inference rule shorthand
  - concretely or constructively where S is the limit of a series of sets
- None of the definitions actually describe the meaning of terms with respect to the values they represent
- Are these different definitional styles equivalent?

Generating functions



- Each inference rule defining T can be thought of as a generating function that given some elements from T, generates new elements of T.
- To say T is closed under these rules means that T cannot be made bigger using these generating functions.

Generating functions



F1(U) = {`true`}

F2(U) = {`false`}

F3(U) = {`0`}

F4(U) = {`succ t1` | `t1` ∈ U}

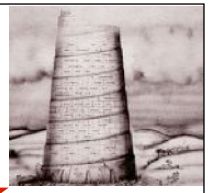
F5(U) = {`pred t1` | `t1` ∈ U}

F6(U) = {`iszero t1` | `t1` ∈ U}

F7(U) = {**`if`** `t1` **`then`** `t2` **`else`** `t3` | `t1,t2,t3` ∈ U}

Each function takes a set of terms U as input and produces a set of terms “justified by U” as output

Relating back to T

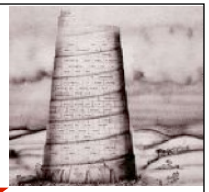


- We now define

$$F(U) = \bigcup_i F(U)$$

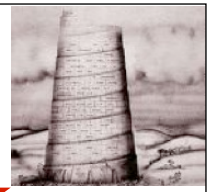
- Definition:
  - A set U is said to be closed under F (or F-closed) if F(U) ⊆ U
  - The set of terms T is the smallest F-closed set

Relating back to S



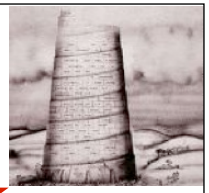
- We now have two constructive definitions that characterize the same set from different directions:
  - “from above” as the intersection of all F-closed sets
  - “from below” as the limit (union) of a series of sets that start from {} and “get closer” to being F-closed

T = S



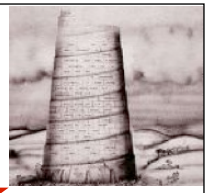
- Proof: T is defined as the smallest set satisfying certain conditions. Suffice to show that (a) S satisfies these conditions and (b) any set satisfying these conditions has S as a subset
- Can prove (a) by inspection
- Can prove (b) by complete induction on i.
  - Suppose S' satisfying the three conditions defining T. Show that for any i, $S_i \subseteq S'$, thus implying $S \subseteq S'$.
    Assume $S_j \subseteq S'$ for $j < i$ and show that $S_i \subseteq S'$.

Induction on Terms



- if `t` ∈ T then
  - `t` is a constant
  - `t` is of the form `succ t'`, `pred t'`, `iszero t'` for some smaller term `t'`
  - `t` is of the form `if t1 then t2 else t3` for some smaller terms `t1, t2, t3`
- Can apply this observation to
  - define inductive definitions of functions over terms
  - inductive proofs of properties over terms

Example



- The set of constants appearing in a term `t` written Consts(`t`) is defined as:

  Consts(`true`) = { `true` }

  Consts(`false`) = {`false`}

  Consts(`0`) = {`0`}

  Consts(`succ t`) = Consts(`t`)

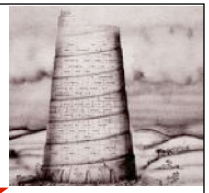
  Consts(`pred t`) = Consts(`t`)

  Consts(`iszero t`) = Consts(`t`)

  Consts(`if t1 then t2 else t3`) =

  Consts(`t1`) ∪ Consts(`t2`) ∪ Consts(`t3`)

Inductive definitions

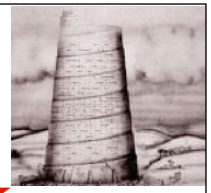


- In what sense is this a definition?
  - The thing we are defining is defined in terms of the thing we are defining
  - But, the specification has the essential trait of being unambiguous (it defines a function):

    total: every element in the range is related by at least one element in its domain

    deterministic: every element in the domain is related to at most one element in its range

Inductive definition



- An alternative formulation:

BadConsts(`true`) = { `true` }

BadConsts(`false`) = {`false`}

BadConsts(`0`) = {`0`}

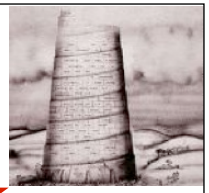
BadConsts(`0`) = {}

BadConsts(`succ t`) = BadConsts(`t`)

BadConsts(`pred t`) = BadConsts(`t`)

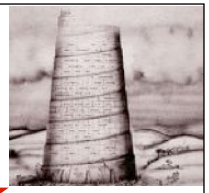
BadConsts(`iszero t`) = BadConsts(`iszero (iszero t)`)

Inductive definitions



- BadConsts is not a well-formed inductive definition:
  - it is not deterministic (two rules for `0`)
  - it is not total (no rule for `if`)
  - it is not inductive (rule for `iszero`)

Another inductive definition



Size(`true`) = 1

Size(`false`) = 1

Size(`0`) = 1

Size(`succ t1`) = Size(t1) + 1

Size(`pred t1`) = Size(t1) + 1

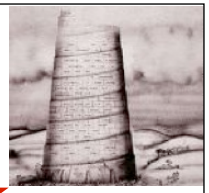
Size(`iszero t1`) = Size(t1) + 1

Size(`if t1 then t2 else t3`) =

Size(`t1`) + Size(`t2`) + Size(`t3`) + 1

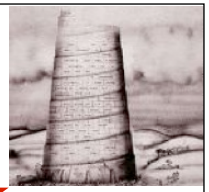
- The depth of a term t is the smallest i such that $t \in S_i$

Inductive proofs on terms



- Lemma: The number of distinct constants in term `t` is no greater than the size of t (i.e., |Consts(`t`)| ≤ Size(`t`))
- Proof: By induction on the depth of `t`.
- Assuming the desired property for all terms of smaller depth than `t` holds, we must prove it for `t` itself.

Inductive proofs on terms



- Case: `t` is a constant

  | Consts(`t`) | = |{`t`}| = 1 = Size(`t`)

- Case: `t` = `succ(t1)`, `pred(t1)`, `iszero(t1)`

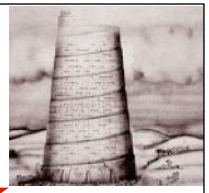
  By the induction hypothesis,

  |Consts(`t1`) | ≤ Size(`t1`).

  Now, |Consts(`t`)| = |Consts(`t1`)| ·

  Size(`t1`) < Size(`t`)

Inductive proofs on terms



- Case: `t = if t1 then t2 else t3`

  By the induction hypothesis,

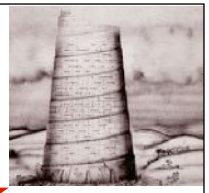
  $|\text{Consts}(t_i)| \leq \text{Size}(t_i)$, $1 \leq i \leq 3$.

  Now, $|\text{Consts}(t)| = |\text{Consts}(t1) \cup \text{Consts}(t2) \cup \text{Consts}(t3)|$

  $\leq |\text{Consts}(t1)| + |\text{Consts}(t2)| + |\text{Consts}(t3)|$

  $\leq \text{Size}(t1) + \text{Size}(t2) + \text{Size}(t3) < \text{Size}(t)$

Structural induction



- If for each term `s`, given P(`r`) for all immediate subterms `r` of `s`, we can show P(`s`), then P(`s`) holds for all `s`.

- Variants:
  - Induction by depth:

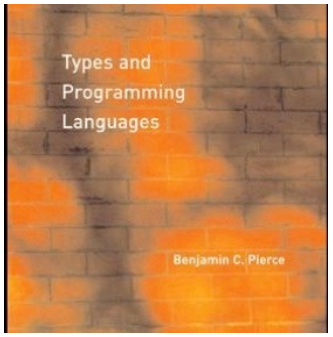
    If for each term s, given P(`r`) for all `r` such that depth(`r`) < depth(`s`), we can show P(`s`), then P(`s`) holds for all `s`.
  - Induction on size:

    If for each term `s`, given P(`r`) for all r such that size(`r`) < size (`s`), we can show P(`s`), then P(`s`) holds for all `s`.

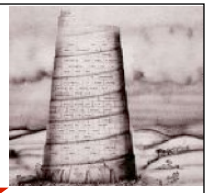
Operational Semantics

CS 565

Lecture 3



Review



- Abstract syntax trees
  - view as parse tree for a program

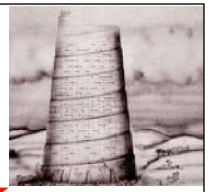
    independent of concrete syntax

    does not provide a semantics for operators
  - BNF grammars and related inductive definition styles allow us to:

    specify certain structural properties of programs (e.g., size, depth, etc.) without knowledge of their semantics

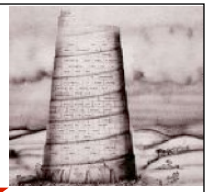
    write inductive-style proofs that relate these properties

Semantics



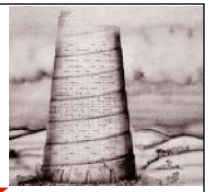
- We are ultimately interested in the meaning of programs:
  - How do we define “meaning”?
  - How do we understand notions like “evaluation”, “compilation”?
  - How is “evaluation” related to “meaning”?
  - How do we capture notions like “non-termination”, “recursion”, etc. in defining the “meaning” of a program?

Abstract machines



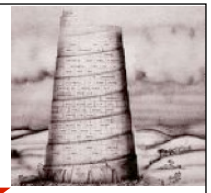
- First approach: define an abstract machine.
- The behavior of the machine on a program defines the program's "meaning".
- An abstract machine consists of:
  - a set of states
  - a transition relation on states (→)
- Evaluation stops when we reach a state in which no further transitions are possible.

States and Transitions



- States record all salient information in a machine:
  - program counter
  - register contents
  - memory
  - code
- In studying languages, we can abstract these complex low-level structures to simpler high-level ones
  - For the simple language of arithmetic, the state is simply the term being evaluated
- The transition relation is often a partial function on states:
  - Not all states have a transition
- If a state does have a transition, the resulting state is unique

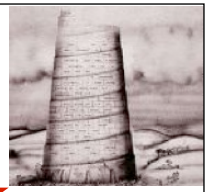
Booleans



- Syntax of terms and values

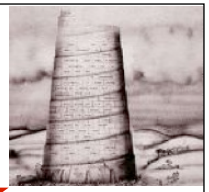
t	$::=$		
	$\mid$	$\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t''$	conditional
	$\mid$	$\texttt{true}$	true constant
	$\mid$	$\texttt{false}$	false constant
v	$::=$		
	$\mid$	$\mathbf{tt}$	true value
	$\mid$	$\mathbf{ff}$	false value

Transition (Evaluation) Relation



- The relation $t \rightarrow t'$ is the smallest relation closed under the following rules:

Terminology

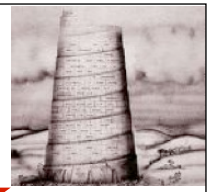


- Computation rules

- Congruence rule

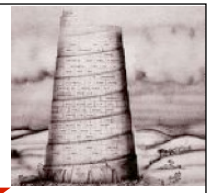
- Computation rules perform “real” computation steps.
- Congruence rules guide evaluation order; they determine where computation rules can be next applied

Example



- Consider a different evaluation strategy such that
  - the then and else branches are evaluated (in that order) before the guard and
  - if the then and else branches both yield the same value, we omit evaluation of the guard.
- How would we write this evaluator?

An alternative evaluator



if true then vt else vf → vt

if false then vt else vf → vf

if t1 then v else v → v

$\boxed{t_1 \longrightarrow t_2}$

$$\frac{}{\textbf{if } \texttt{true} \textbf{ then } t_1 \textbf{ else } t_2 \longrightarrow t_1} \quad \textsc{Rtrue}$$

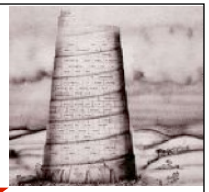
$$\frac{}{\textbf{if } \texttt{false} \textbf{ then } t_1 \textbf{ else } t_2 \longrightarrow t_2} \quad \textsc{Rfalse}$$

$$\frac{t_1 \longrightarrow t'}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \longrightarrow \textbf{if } t' \textbf{ then } t_2 \textbf{ else } t_3} \quad \textsc{RredC}$$

$$\frac{t_2 \longrightarrow t'}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \longrightarrow \textbf{if } t_1 \textbf{ then } t' \textbf{ else } t_3} \quad \textsc{RredT}$$

$$\frac{t_3 \longrightarrow t'}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \longrightarrow \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t'} \quad \textsc{RredF}$$

Induction

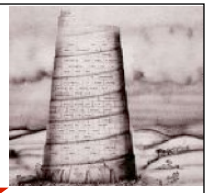


- We view the transition relation as the smallest binary relation on terms satisfying the rules. If $(t,t') \in \rightarrow$, then the judgment $t \rightarrow t'$ is derivable.

- A derivation tree is a tree whose leaves are instances of computation rules (e.g., true and false transitions) and whose internal nodes are congruence rules.

- This notion of evaluation as a construction of a tree leads to an inductive proof technique on induction on derivations.

Derivation trees



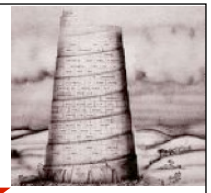
- Consider the following terms:

$$\mathbf{s} \equiv \texttt{if true then false else false}$$
$$\mathbf{t} \equiv \texttt{if s then true else true}$$
$$\mathbf{u} \equiv \texttt{if false then true else true}$$

- What is the derivation tree for the judgment?

$$\texttt{if } \mathbf{t} \texttt{ then false else false} \rightarrow \texttt{if } \mathbf{u} \texttt{ then false else false}$$

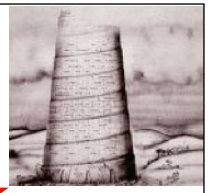
Derivation Trees



$$\dfrac{\dfrac{}{s \rightarrow \texttt{false}}}{t \rightarrow u}$$

$$\texttt{if } t \texttt{ then false else false} \rightarrow \texttt{if } u \texttt{ then false else false}$$

Induction

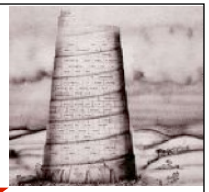


- Theorem: if $t \rightarrow t'$ and $t \rightarrow t''$ then $t' = t''$.

- Proof: By induction on the derivation of $t \rightarrow t'$. At each step of the induction, assume theorem holds for all smaller derivations. Proceed by case analysis of the evaluation rule used at the root of the derivation.

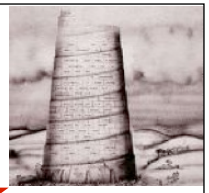
- Theorem: if $t \rightarrow t'$ then size(t) > size(t')

Normal forms



- A term t is in normal form if no evaluation rule applies to it, i.e., there is no `t'` such that $t \longrightarrow t'$.
- Every value is in normal form.
- Theorem: Every term that is in normal form is a value.
  - Proof: How would you prove this?

Normal forms

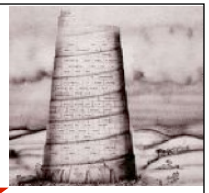


- **Theorem**: Every term `t` that is in normal form is a value.*
- **Proof**:
  - By structural induction on `t` and contradiction.
  - Suppose `t` is not a value.
  - `t` must have the form "`if t1 then t2 else t3`"

    Now, `t1` can be either `true` or `false` in which case `t` is not in normal form (there is a computation rule that matches), or `t1` is another if expression.

    By the induction hypothesis (*), `t1` is not in normal form, hence `t` is not in normal form.

Normal forms



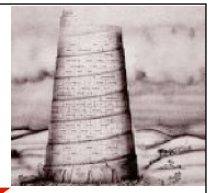
- Is it always the case for real languages that a term which is in normal form is always a value?
  - In real languages normal forms may also correspond to expressions that are ill-typed or which correspond to runtime errors.

    E.g., `true + 3` → ??? or

    `succ false` → ?

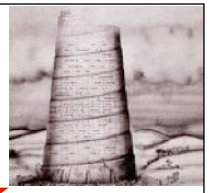
    These terms are in normal form (why?) but do not correspond to values as defined by the machine specification.
  - A term is said to be stuck if it is normal form but is not a value

IMP: A simple imperative language



- Syntactic categories:
  - int — Integers
  - bool — Boolean
  - loc — Locations
  - Aexp — Arithmetic expressions
  - Bexp — Boolean expressions
  - Com — Commands
  - Values — v ::= n | true | false

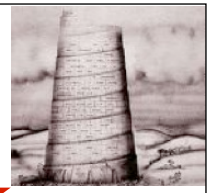
Abstract syntax (AExp)



- Arithmetic expressions:
  - Variables are used directly in expressions (no prior declaration)
  - All variables are presumed to have integer type
  - No side-effects (e.g., overflow, etc.)

$$
\begin{array}{lll}
a & ::= & \\
 & | & int \\
 & | & \mathbf{x} \\
 & | & a_1 + a_2 \\
 & | & a_1 * a_2 \\
 & | & a_1 - a_2
\end{array}
$$

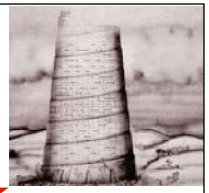
Abstract Syntax (BExp)



- Boolean expressions:

$$
\begin{array}{lll}
b & ::= & \\
 & | & \mathit{bool} \\
 & | & e_1 = e_2 \\
 & | & e_1 \prec e_2 \\
 & | & \mathbf{not}\ b \\
 & | & b_1\ \mathbf{and}\ b_2 \\
 & | & b_1\ \mathbf{or}\ b_2
\end{array}
$$

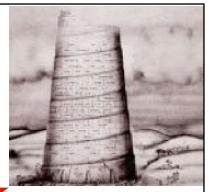
Abstract syntax (Comm)



- Commands
  - Typing rules expressed implicitly in the choice of meta-variables
  - All side-effects captured within commands
  - Do not consider functions, pointers, data structures

$$
\begin{array}{lcl}
c & ::= & \\
 & | & \mathbf{skip} \\
 & | & \mathbf{x} := a \\
 & | & c_1 \,;\; c_2 \\
 & | & \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\
 & | & \mathbf{while}\ b\ \mathbf{do}\ c
\end{array}
$$

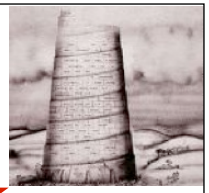
Operational Semantics for IMP



- Unlike the simple language of booleans and conditionals or arithmetic, IMP programs bind variables to locations, and can side-effect the contents of these locations.
- Define $\sigma \in \Sigma = L \rightarrow Z$ to define the state of program memory.
- Evaluation judgements take one of the following forms:
  - $c, \sigma \rightarrow c', \sigma'$
  - $e, \sigma \rightarrow e'$

    $e \in exp = Aexp + Bexp + Value$

Semantics for Aexp



- Notes
  - σ does not change; because aexps do not have side-effects
  - distinctions between normal forms (values) and expressions expressed in the choice of meta-variables used in the rules
  - order of evaluation expressed in the definition of the rules

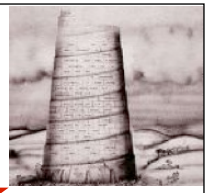
$$\frac{\sigma\,(\mathbf{x}) = int}{\mathbf{x},\,\sigma \longrightarrow int} \quad \textsc{AexpVar}$$

$$\frac{a_1,\,\sigma \longrightarrow a_1'}{a_1 + a_2,\,\sigma \longrightarrow a_1' + a_2} \quad \textsc{AexpPlusL}$$

$$\frac{a_2,\,\sigma \longrightarrow a_2'}{int + a_2,\,\sigma \longrightarrow int + a_2'} \quad \textsc{AexpPlusR}$$

$$\frac{int_1 + int_2 = int_3}{int_1 + int_2,\,\sigma \longrightarrow int_3} \quad \textsc{AexpPlus}$$

Semantics for Aexp



$$\frac{a_1,\ \sigma \longrightarrow a_1'}{a_1 * a_2,\ \sigma \longrightarrow a_1' * a_2} \quad \textsc{AexpTimesL}$$

$$\frac{a_2,\ \sigma \longrightarrow a_2'}{int * a_2,\ \sigma \longrightarrow int * a_2'} \quad \textsc{AexpTimesR}$$

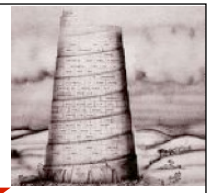
$$\frac{int_1 * int_2 = int_3}{int_1 * int_2,\ \sigma \longrightarrow int_3} \quad \textsc{AexpTimes}$$

$$\frac{a_1,\ \sigma \longrightarrow a_1'}{a_1 - a_2,\ \sigma \longrightarrow a_1' - a_2} \quad \textsc{AexpSubL}$$

$$\frac{a_2,\ \sigma \longrightarrow a_2'}{int - a_2,\ \sigma \longrightarrow int - a_2'} \quad \textsc{AexpSubR}$$

$$\frac{int_1 - int_2 = int_3}{int_1 - int_2,\ \sigma \longrightarrow int_3} \quad \textsc{AexpSub}$$

Semantics for BExp



$$\frac{e_2,\ \sigma \longrightarrow e_2'}{int = e_2,\ \sigma \longrightarrow int = e_2'} \quad \textsc{BexpEqR}$$

$$\frac{}{\textbf{not}\ \texttt{true},\ \sigma \longrightarrow \texttt{false}} \quad \textsc{BexpNotT}$$

$$\frac{}{\textbf{not}\ \texttt{false},\ \sigma \longrightarrow \texttt{true}} \quad \textsc{BexpNotF}$$

$$\frac{b,\ \sigma \longrightarrow b'}{\textbf{not}\ b,\ \sigma \longrightarrow \textbf{not}\ b'} \quad \textsc{BexpNot}$$

$$\frac{bool_1\ \textbf{and}\ bool_2 = bool}{bool_1\ \textbf{and}\ bool_2,\ \sigma \longrightarrow bool} \quad \textsc{BexpAnd}$$

$$\frac{b_1,\ \sigma \longrightarrow b_1'}{b_1\ \textbf{and}\ b_2,\ \sigma \longrightarrow b_1'\ \textbf{and}\ b_2} \quad \textsc{BexpAndL}$$

$$\frac{b_2,\ \sigma \longrightarrow b_2'}{bool\ \textbf{and}\ b_2,\ \sigma \longrightarrow bool\ \textbf{and}\ b_2'} \quad \textsc{BexpAndR}$$

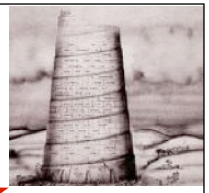
$$\frac{bool_1\ \textbf{or}\ bool_2 = bool}{bool_1\ \textbf{or}\ bool_2,\ \sigma \longrightarrow bool} \quad \textsc{BexpOr}$$

$$\frac{b_1,\ \sigma \longrightarrow b_1'}{b_1\ \textbf{or}\ b_2,\ \sigma \longrightarrow b_1'\ \textbf{or}\ b_2} \quad \textsc{BexpOrL}$$

$$\frac{b_2,\ \sigma \longrightarrow b_2'}{bool\ \textbf{or}\ b_2,\ \sigma \longrightarrow bool\ \textbf{or}\ b_2'} \quad \textsc{BexpOrR}$$

v==> int

Semantics for Com



$$\frac{}{\textbf{skip}\,;\ c,\ \sigma \longrightarrow c,\ \sigma} \quad \textsc{Skip}$$

$$\frac{a,\ \sigma \longrightarrow a'}{\mathbf{x} := a\,;\ c,\ \sigma \longrightarrow \mathbf{x} := a'\,;\ c,\ \sigma} \quad \textsc{AssignE}$$

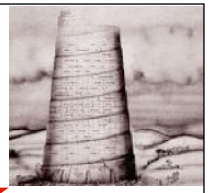
$$\frac{\sigma' = \sigma\,[\mathbf{x} \mapsto int]}{\mathbf{x} := int\,;\ c,\ \sigma \longrightarrow c,\ \sigma'} \quad \textsc{Assign}$$

$$\frac{b,\ \sigma \longrightarrow b'}{b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\,;\ c_3,\ \sigma \longrightarrow \textbf{if}\ b'\ \textbf{then}\ c_1\ \textbf{else}\ c_2\,;\ c_3,\ \sigma} \quad \textsc{IfE}$$

$$\frac{}{\textbf{if}\ \texttt{true}\ \textbf{then}\ c_1\ \textbf{else}\ c_2\,;\ c_3,\ \sigma \longrightarrow c_1\,;\ c_3,\ \sigma} \quad \textsc{IfT}$$

$$\frac{}{\textbf{if}\ \texttt{false}\ \textbf{then}\ c_1\ \textbf{else}\ c_2\,;\ c_3,\ \sigma \longrightarrow c_2\,;\ c_3,\ \sigma} \quad \textsc{IfF}$$

Semantics for Com

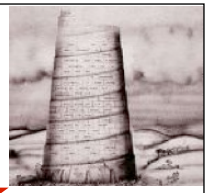


$$\frac{b,\ \sigma \longrightarrow b'}{\textbf{while}\ b\ \textbf{do}\ c_1\ ;\ c_2,\ \sigma \longrightarrow \textbf{while}\ b'\ \textbf{do}\ c_1\ ;\ c_2,\ \sigma}\ \textsc{WhileE}$$

$$\frac{}{\textbf{while}\ \texttt{true}\ \textbf{do}\ c_1\ ;\ c_2,\ \sigma \longrightarrow c_1\ ;\ \textbf{while}\ \texttt{true}\ \textbf{do}\ c_1\ ;\ c_2,\ \sigma}\ \textsc{WhileT}$$

$$\frac{}{\textbf{while}\ \texttt{false}\ \textbf{do}\ c_1\ ;\ c_2,\ \sigma \longrightarrow c_2,\ \sigma}\ \textsc{WhileF}$$

Semantics for Com



- There are some issues with the Com rules:
  - There are many “uninteresting” rules that merely reduce subexpressions
  - All programs must be terminated by `skip`
  - What happens in the `While` rules if `b` depends on state modified by `c1`? (Think of a fix)