Imperative Languages: Informal Description

INFORMAL SEMANTICS CAN BE IMPRECISE

What is the meaning of the C expression f(i++, --i)?

- I++ increases the value of i by one and returns the old value of i
- --I decreases the value of I by one and return the new value of i

The order of evaluation of parameters of functions is not specified in the definition of Language C. Therefore, different C compilers may generate very different code for f(i++, --i)

Imperative Languages: Formal Description

The first approach to giving a precise, formal description of the behaviour of programming constructs was in terms of an ***abstract machine.***

AN ABSTRACT MACHINE IS A TRANSITION SYSTEM, WHICH SPECIFIES AN INTERPRETER FOR THE PROGRAMMING LANGUAGE.

- We will give an example of this for a small ***imperative language*: SIMP**

Abstract Syntax of SIMP

Programs

$$P ::= C \mid E \mid B$$

Commands

$$C ::= skip \mid I := E \mid C; C \mid if\ B\ then\ C\ else\ C \mid while\ B\ do\ C$$

Integer Expressions

$$E ::= !I \mid n \mid E\ op\ E \qquad op ::= + \mid - \mid * \mid /$$

where
- $n \in Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (integers)
- $I \in L = \{l_0, l_1, \ldots\}$ (locations or variables). The expression $!I$ denotes the value stored in I.

Boolean Expressions

$$B ::= True \mid False \mid E\ bop\ E \mid \neg B \mid B \wedge B$$

$$bop ::= > \mid < \mid =$$

- This grammar describes trees

`

- " L := E " : label (name of variable) is assigned to Expression E (E is non-terminal)

- " C;C " : two commands as a sequence delimited by ' ; ' . it means a tree with a root node as a sequence token, with two subtrees for two commands *C1* and *C2.*

- "if B then C else C" tree which have a root which is a two-way selector with 3 subtrees, one for Boolean condition, one for the commands that need to be executed if condition is true, and one for commands that need to be executed if condition is false.

- " while B do C " syntax tree that contains root saying it's a loop (can be called while) with two subtrees: one for conditions B and one for commands C.
- " E bop E = comparison expression
- " skip " is there for convince and does not do anything however when for example the we are obligated to always put else in the two-way selector and we only want to check for B then we can put ELSE skip. To return to the point before execution.
- !L = the content of the cell associated to L. L = address of the cell.

Examples of program in SIMP

***Swapping the contents of the variables** x **and** y:*

z := !x ; x := !y ; y := !z

This is ambiguous because there are two possible abstract syntax trees. BECAUSE GRAMMAR GIVEN TO US SPECIFIES THAT SEQUENCE have two subtrees. We don't know which one to select for a root.

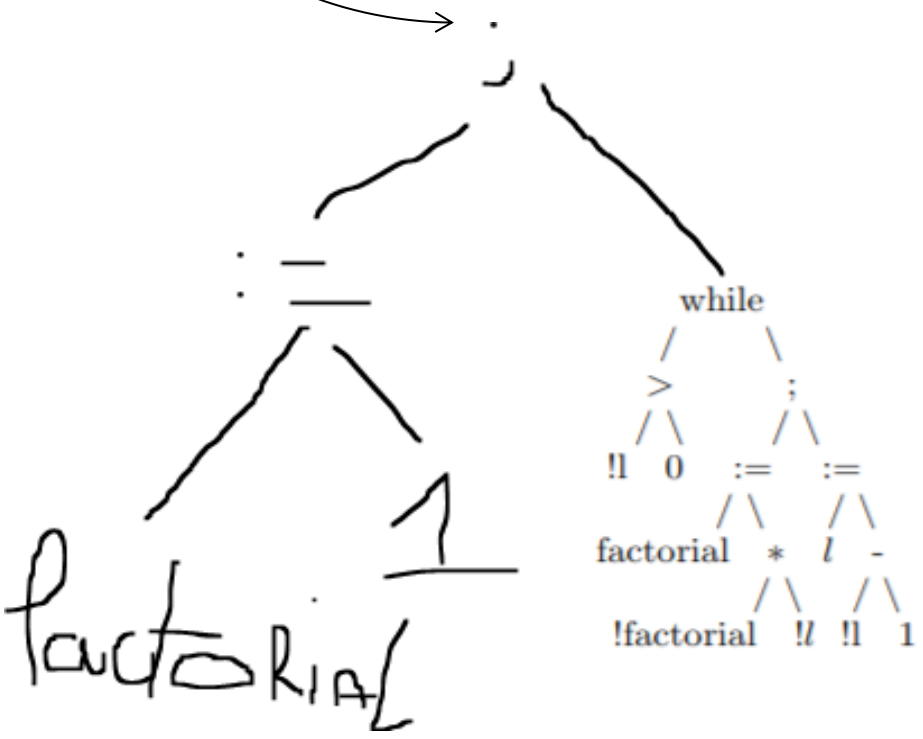
CAN ADD BRACKETS TO RESOLVE THE AMBIGIOUTY

Factorial:

Assuming the variable ' l ' contains a natural number ' n ', the following program computes ' n! '

```
Factorial := 1; while !l > 0 do
                    (factorial := !factorial * !l; l := !l -1)
```

Example – abstract syntax tree



An abstract Machine for SIMP

WE WILL DEIFNE AN *ABSTRACT MACHINE* WITH:

- **CONTROL STACK = this is where we put all commands**
- **AUXILARY STACK = intermediate memory.**

We will define an *abstract machine* with

1. a control stack c
2. an auxiliary stack r (also called results stack)
3. memory, also called store, modelled by a partial function m, mapping each address to an integer.
   $dom(m)$ denotes the locations where m is defined.

   Notation: $m[l \mapsto n]$ is the function that maps each $l' \neq l$ to the value $m(l')$, and l to the value n.
   More precisely:
   $m[l \mapsto n](l) = n$
   $m[l \mapsto n](l') = m(l') \quad$ if $l \neq l'$

The abstract machine is defined by a set of configurations and a set of transition rules.

- M(l) get me value from memory cell at address l;
- You give an address : " L " and M outputs the value that it has at this address, m(x) we want the content that is stored at the address X.
- This notation here describes that memory location (L) was updated with value N and if we want to return the value stored at memory location L it will be N.

Configurations

A CONFIGURATION IS A TRIPLE: <CONTROL STACK, RESULT STACK, MEMORY>

Definition of the control and results stacks:

$$c ::= nil \mid i \cdot c$$
$$i ::= P \mid op \mid \neg \mid \wedge \mid bop \mid := \mid if \mid while$$
$$r ::= nil \mid P \cdot r \mid l \cdot r$$

where P, bop and op are the non-terminals used in the rules defining SIMP programs.

To model the execution of a program we will define transitions between initial and final configurations.
Initial Configurations: $\langle C \cdot nil, nil, m \rangle$
Final Configurations: $\langle nil, nil, m \rangle$

- C = can be either empty notated with ' **NIL** ' if they are not empty stack is denoted as a conjunction of element and a control stack. **I * c some element and the rest. (SYNTAX FOR STACKS)**

- i ::= (ELEMENTS THAT WE CAN PUT ON THE CONTROL STACK)
  - P (PROGRAM) – is an abstract syntax tree is a command, expression or Boolean expression.
  - OP, NOT, AND, BOP (COMPARISON OPERATOR), := , IF, WHILE

- r = Result stack can be either empty or contain program or a label.
- When we start machine result stack is empty, memory can be anything.

Week 3 Week 4

Semantics of SIMP Programs

WE NOW NEED TO DEFINE THE COMPUTATION STEPS

- If there is a sequence of transitions

$$(C \cdot nil, nil, m) \longrightarrow^* (nil, nil, m')$$

then we say that the program C executed in the state m terminates successfully producing the state m'.
The *semantics of a command C in the state m* is described by giving the sequence of *transitions* that transform the configuration $(C \cdot nil, nil, m)$ into (nil, nil, m').

- If there is a sequence of transitions

$$(E \cdot c, r, m) \longrightarrow^* (c, v \cdot r, m')$$

$$(B \cdot c, r, m) \longrightarrow^* (c, v \cdot r, m')$$

then we say that the value of the expression E (resp. B) in the state m is v.

Transition Rules

Evaluation of Expressions

- For example, when we are in state where we have a token N on the control stack, after one step of computation we will end up with control stack and N value in the result stack.
- Same for Boolean
- Negated B is popped of the control stack and tokenized into separate NOT and separate B and then placed on the control stack. Where another transition rule can pop the not pop the B and then push the negation of B onto the result stack. So basically, the first thing will be to separate not and a b, then to evaluate the b and then to negate it and push it onto the result stack.

$$\begin{aligned}
\langle n \cdot c, r, m\rangle &\rightarrow \langle c, n \cdot r, m\rangle \\
\langle b \cdot c, r, m\rangle &\rightarrow \langle c, b \cdot r, m\rangle \\
\langle \neg B \cdot c, r, m\rangle &\rightarrow \langle B \cdot \neg \cdot c, r, m\rangle \\
\langle (B_1 \wedge B_2) \cdot c, r, m\rangle &\rightarrow \langle B_1 \cdot B_2 \cdot \wedge \cdot c, r, m\rangle \\
\langle \neg \cdot c, b \cdot r, m\rangle &\rightarrow \langle c, b' \cdot r, m\rangle \quad \text{if } b' = not\ b \\
\langle \wedge \cdot c, b_2 \cdot b_1 \cdot r, m\rangle &\rightarrow \langle c, b \cdot r, m\rangle \quad \text{if } b_1 \text{ and } b_2 = b \\
\langle (E_1\ op\ E_2) \cdot c, r, m\rangle &\rightarrow \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle \\
\langle (E_1\ bop\ E_2) \cdot c, r, m\rangle &\rightarrow \langle E_1 \cdot E_2 \cdot bop \cdot c, r, m\rangle \\
\langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle &\rightarrow \langle c, n \cdot r, m\rangle \quad \text{if } n_1\ op\ n_2 = n \\
\langle bop \cdot c, n_2 \cdot n_1 \cdot r, m\rangle &\rightarrow \langle c, b \cdot r, m\rangle \quad \text{if } n_1\ bop\ n_2 = b \\
\langle !l \cdot c, r, m\rangle &\rightarrow \langle c, n \cdot r, m\rangle \quad \text{if } m(l) = n
\end{aligned}$$

- For example, E1 op E2 will be split and placed on the result stack separately, but first they will be tokenized into simple forms. It is important to push OP or BOP first but the order of E1 and E2 is irrelevant.

Evaluation of Commands

- For example, when we are assigning L to E, E gets pushed onto the control stack separately and this is because we must evaluate E first before placing it in the location L. WE DON'T KNOW YET IF IT'S A NUMBER OPERATION OR SOMETHING ELSE. AND WE PUT THE ' L ' ON TOP OF THE RESULT STACK BECAUASE WE NEED TO REMEMBER THE LOCATION OF ASSIGNMENT.
- If command starts from pushing tokens onto the control stack, with the B on top, because its popped as first node to check the condition, if the condition returns true, load C1 else load C2
- While condition loads the B onto the stack to evaluate it if it is true then execute the C command and put the while loop command next in the control stack for the effect of repetition. IF B returns false it means the loop has terminated.

$$\begin{aligned}
\langle skip \cdot c, r, m\rangle &\rightarrow \langle c, r, m\rangle \\
\langle (l := E) \cdot c, r, m\rangle &\rightarrow \langle E \cdot := \cdot c, l \cdot r, m\rangle \\
\langle := \cdot c, n \cdot l \cdot r, m\rangle &\rightarrow \langle c, r, m[l \mapsto n]\rangle \\
\langle (C_1; C_2) \cdot c, r, m\rangle &\rightarrow \langle C_1 \cdot C_2 \cdot c, r, m\rangle \\
\\
\langle (if\ B\ then\ C_1\ else\ C_2) \cdot c, r, m\rangle &\rightarrow \langle B \cdot if \cdot c, C_1 \cdot C_2 \cdot r, m\rangle \\
\langle if \cdot c, True \cdot C_1 \cdot C_2 \cdot r, m\rangle &\rightarrow \langle C_1 \cdot c, r, m\rangle \\
\langle if \cdot c, False \cdot C_1 \cdot C_2 \cdot r, m\rangle &\rightarrow \langle C_2 \cdot c, r, m\rangle \\
\\
\langle (while\ B\ do\ C) \cdot c, r, m\rangle &\rightarrow \langle B \cdot while \cdot c, B \cdot C \cdot r, m\rangle \\
\langle while \cdot c, True \cdot B \cdot C \cdot r, m\rangle &\rightarrow \langle C \cdot (while\ B\ do\ C) \cdot c, r, m\rangle \\
\langle while \cdot c, False \cdot B \cdot C \cdot r, m\rangle &\rightarrow \langle c, r, m\rangle
\end{aligned}$$

Examples of Transitions

Let $C = \mathit{while}\ B\ \mathit{do}\ C'$
and $B = !I > 0$
and $C' = \mathit{factorial} := !\mathit{factorial} * !I;\ I := !I - 1$
and $m = \{I \mapsto 4, \mathit{factorial} \mapsto 1\}$.

$\langle C \cdot nil, nil, m\rangle \to \langle B \cdot while \cdot nil, B \cdot C' \cdot nil, m\rangle$
$\to \langle !I \cdot 0 \cdot > \cdot while \cdot nil, B \cdot C' \cdot nil, m\rangle$
$\to \langle 0 \cdot > \cdot while \cdot nil, 4 \cdot B \cdot C' \cdot nil, m\rangle$
$\to \langle > \cdot while \cdot nil, 0 \cdot 4 \cdot B \cdot C' \cdot nil, m\rangle$
$\to \langle while \cdot nil, True \cdot B \cdot C' \cdot nil, m\rangle$
$\to \langle C' \cdot C \cdot nil, nil, m\rangle$
...
$\to \langle nil, nil, m[I \mapsto 0, \mathit{factorial} \mapsto 24]\rangle$

The abstract machine describes the exact behaviour of a while loop. Compare with the informal definition: *... the body of the loop is executed as long as the value of the expression is true.*

Operational Semantics

Advantage of the abstract machine: The abstract machine explains the execution of the commands step by step, which is useful if we must implement the language.

Disadvantage of the abstract machine: It is not very intuitive. Many transitions are doing just phrase analysis, only a few really perform computation.

Structural Approach to Operational Semantics

To overcome the previous problem, another approach to operation semantics based on transition system was developed: the structural approach (Plotkin).

IDEA: The transition for a compound statement should be defined in terms of the transitions for its constituent sub-statements.

In other words: the definition should be ***INDUCTIVE.***