Formal Semantics

Formal Semantics

Advantages of formal semantics is that translators can be validated to produce exactly the behavior described in the language definition.

Ambiguities and inconsistencies can be discovered.

Methods

- Operational Semantic
- Denotational semantic
- Axiomatic semantic

Operational semantics

- This method defines a language by describing its actions in terms of the operations of an actual or hypothetical machine. Of course, this requires that the operations of the machine used in the description also be precisely defined, and for this reason a very simple hypothetical machine is often used that bears little resemblance to an actual computer.

Denotational semantics.

- This approach uses mathematical functions on programs and program components to specify semantics.
- Programs are translated into functions about which properties can be proved using the standard mathematical theory of function.

Axiomatic semantics

- This method applies mathematical logic to language definition. Assertions, or predicates, are used to describe desired outcomes and initial assumptions for programs.
- Language constructs are associated with **predicate transformers** that create new assertions out of old ones, reflecting the actions of the construct.

Axiomatic semantics

- These transformers can be used to prove that the desired outcome follows from the initial conditions. Thus, this method of formal semantics is aimed specifically at correctness proofs.

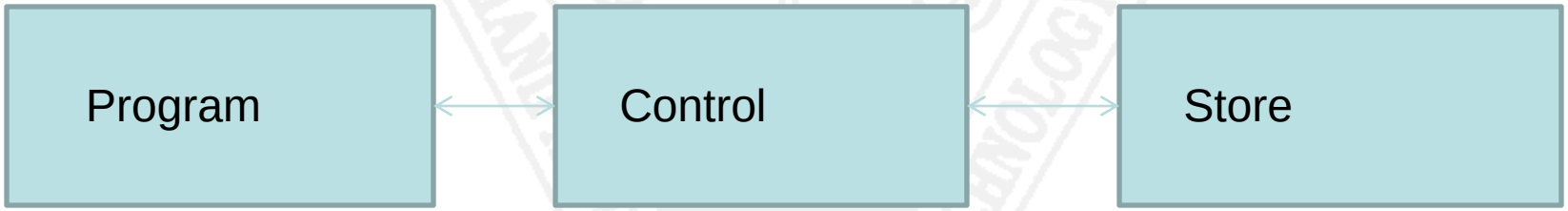
Operational Semantic

- Defines a language by describing its actions in terms of the operations of an actual or hypothetical machine.
- Operations of machine used must also be precisely defined.

Hypothetical machine

- In principle, an abstract machine can be viewed as consisting of three parts: A Program, a control and a store or memory

Hypothetical machine



Hypothetical machine

- We will consider reduction machine whose control operates directly on a program to reduce it to its semantic value

example

- (3+4)*5 =>(7) * 5;3 & 4 are added to get 7
- =>7*5;the parentheses around 7 are dropped.
- =>35 ;7 & 5 are multiplied to get 35.

Semantic Content of a program

- In general, the semantic content of a program will be more than just a numeric value, but as we will see, the semantic content of a program can be represented by a data value of some structured type, which operational semantic reductions will produce.

Logical Inference rules

- Inference rules in logic are written in the following form Premise/ Conclusion.
- That is premise or condition is written first; then a line is drawn, and the conclusion or result is written
- This indicates that whenever premise is true, the conclusion is also true.

Examples

- a+b=c/b+a=c ;commutative property
- a→b,b→c/a→c ;transitive property

Axioms

- Axioms are inference rules with no premise they are always true.
- Eg:- /a+0=a
- Axioms are usually written without a line

Reduction rules for integer arithmetic expresssions

- $E \rightarrow E_1$ ‘+’ E_2 | E_1 ‘-’ E_2 | E_1 ‘*’ E_2 | ‘(‘ E_1’)’
- $N \rightarrow N_1$ D | D
- D→’0’ | ‘1’ | …….|’9’

Conventions

- E, E_1 and so on are used to denote expressions that have not yet been reduced to values; V, V_1 and so on stand for integer values.
- E=>E_1 states that expression E reduces to E_1 by some reduction rule.

Rule1:reduces digits to values

- ‘0’=>0
- ‘1’=>1
- .
- .
- .
- ‘9’=>9

Rule2 Reduces numbers to values

- V ‘0’ => 10 * V
- V ‘1’=> 10* V + 1
- .
- .
- .
- .
- V ‘9’=>10 * V + 9

Rule 3

- V_1 ‘+’ V_2 => $V_1 + V_2$

Rule 4

- V_1 '-' V_2 => $V_1 - V_2$

Rule5

- V_1 ‘*’ V_2 => V_1 * V_2

Rule6

- ‘(‘ V ‘)’ => V

Rule7

- $E => E_1$/E ‘+’ $E_2 => E_1$’+’ E_2

Rule8

- $E => E_1 / E$ ‘-’ $E_2 => E_1$ ’-’ E_2

Rule9

- $E => E_1 / E$ ‘*’ $E_2 => E_1$ ’*’ E_2

Rule 10

- $E => E_1/V$ ‘+’ $E => V$ ‘+’ E_1

Rule11

- $E \Rightarrow E_1 / V$ ‘-’ $E \Rightarrow V$ ‘-’ E_1

Rule12

- E=>E_1/V ‘*’ E => V ‘*’ E_1

Rule 13

- E=>E_1/”(‘ E ‘)’=>’(‘ E_1 ‘)’

Rule14

- $E \Rightarrow E_1$, $E_1 \Rightarrow E_2 / E \Rightarrow E_2$

Explanation

- Rules 1 through 6 are all axioms and rest are inferences.
- Rule 1 says that character ‘0’ (a syntactic entity) reduces to the value 0 (a semantic entity).

Explanation

- Rules 3, 4 and 5 say that whenever we have an expression that consists of two values and an operator symbol, we can reduce that expression to a value by applying the appropriate operation whose symbol appears in the expression.

Explanation

- Rule 6 says that if an expression consists of pair of parentheses surrounding a values, then the parentheses can be dropped.

Explanation

- Rules 7,8 & 9 express the fact that in an expression that consists of an operation applied to other expressions, the left sub-expression may be reduced by itself and that reduction substituted into the larger expression.

Explanation

- Rules 10,11 and 12 express the fact that once a value obtained for left sub-expressions the right sub expression may be reduced.

Explanation

- Rule 13 says that we can first reduce the inside of an expression consisting of parentheses surrounding another expression.

Explanation

- Rule 14 expresses the general fact that reductions can be performed stepwise.

2*(3+4)-5

- We first reduce the 3+4 as follows
- ‘3’ ‘+’ ‘4’=>3 ‘+’ ‘4’ (rule 1 and 7)

  =>3 ‘+’ 4 (rule 1 and 10)

  =>3 + 4 (rule 3)

  =>7 (rule 14)

2*(3+4)-5

- ‘(‘ ‘3‘ ‘+’ ‘4’ ‘)’ => ‘(‘ 7 ‘)’ (rule 13)

  => 7 (rule 6)

2*(3+4)-5

- ‘2’ ‘*’ ‘(‘ ‘3’ ‘+’ ‘4’ ‘)’=> 2 ‘*’ ‘(‘ ‘3’ ‘+’ ‘4’ ‘)’ (rules 1 and 9)

  ⇒2 ‘*’ 7 (rule 12)

  ⇒2 * 7 (rule 14)

  ⇒14 (rule 5)

2*(3+4)-5

- ‘2’ ‘*’ ‘(‘ ‘3’ ‘+’ ‘4’ ‘)’ ‘-’ ‘5’=> 14’-’ ‘5’ Rule 1 and 8
- =>14 ‘-’ 5 rule 11
- =>14-5 rule 4
- =>9

Denotational semantic of Integer Arithmetic expressions

- Syntactic domain
- E:expression
- N: number
- D: digit
- E→E_1 ‘+’ E_2 | E_1 ‘-’ E_2 | ‘(‘ E ‘)’ | N
- N→ N D | D
- D→ ‘0’ | ‘1’ | …….|’9’

Semantic Domains

- Domain V: Integer={….,-2,-1,0,1,2….}
- Operations
- +:IntegerXInteger→Integer
- -:IntegerXInteger→Integer
- *:IntegerXInteger→Integer

Semantic functions

- E:Expression→Integer
- E[[E_1 ‘+’ E_2]]=E[[E_1]]+E[[E_2]]
- E[[E_1 ‘-’ E_2]]=E[[E_1]]-E[[E_2]]
- E[[E_1 ‘*’ E_2]]=E[[E_1]]*E[[E_2]]
- E[[‘(‘ E ’)’]]=E[[E]]
- E[[N]]=N[[N]]

Semantic functions

- N: Number→Integer
- N[[ND]]=10*N[[N]]+N[[D]]
- N[[D]]=D[[D]]
- D:Digit→Integer
- D[[‘0’]]=0, D[[‘1’]]=1, ….D[[‘9’]]=9

E[[‘(‘ ‘2’ ‘+’ ‘3’ ‘)’ ‘*’ ‘4’]]

- = E[[‘(‘ ‘2’ ‘+’ ‘3’ ‘)’]] * E[[‘4’]]
- = E[[‘2’ + ‘3’]] * N[[‘4’]]
- =(E[[‘2’]]+E[[‘3’]])*D[[‘4’]]
- =(N[[‘2’]]+N[[‘3’]])*4
- =(D[[‘2’]]+D[[‘3’]])*4
- =(2+3)*4=5*4=20

Axiomatic Semantics

- Assertion about the situation just before execution are called preconditions and assertion about the situation after execution are called postconditions.

Example

- X=X+1
- X=A before execution(precondition)
- X=A+1 after execution(postcondition)
- {X=A}X=X+1 {X=A+1}

Example

- X=1/Y
- {Y!=0} X=1/Y { X=1/Y}

Axiomatic Specification

- In general axiomatic specification of the semantics of the language construct C is of the form {P} C {Q}
- Where P and Q are assertions. The meaning of such a specification is that,
- If P is true just before the execution of C, then Q is true just after the execution of C.

References

Text book

- Kenneth C. Louden “Programming Languages Principles and Practice” second edition Thomson Brooks/Cole Publication.

Reference Books:

- Terrence W. Pratt, Masvin V. Zelkowitz “Programming Languages design and Implementation” Fourth Edition Pearson Education.
- Allen Tucker, Robert Noonan “Programming Languages Principles and Paradigms second edition Tata MC Graw –Hill Publication.