Semantics

- Static semantics
  - attribute grammars
    - examples
    - computing attribute values
    - status
- Dynamic semantics
  - operational semantics
  - axiomatic semantics
    - examples
    - loop invariants
    - evaluation
  - denotational semantics
    - examples
    - evaluation

Static Semantics

- Used to define things about PLs that are hard or impossible to define with BNF
  - hard: type compatibility
  - impossible: declare before use
- Can be determined at compile time
  - hence the term static
- Often specified using natural language descriptions
  - imprecise
- Better approach is to use attribute grammars
  - Knuth (1968)

Attribute Grammars

- Carry some semantic information along through parse tree
- Useful for
  - static semantic specification
  - static semantic checking in compilers
- An attribute grammar is a CFG G = (S, N, T, P) with the additions
  - for each grammar symbol x there is a set A(x) of attribute values
  - each production rule has a set of functions that define certain attributes of the non-terminals in the rule
  - each production rule has a (possibly empty) set of predicates to check for attribute consistency
    - valid derivations have predicates true for each node

Attribute Grammars (continued)

- Synthesized attributes
  - are determined from nodes of children in parse tree
    - if X0 -> X1 ... Xn is a rule, then S(X0) = f(A(X1), ..., A(Xn))
  - pass semantic information up the tree
- Inherited attributes
  - are determined from parent and siblings
    - I(Xj) = f(A(X0), ..., A(Xn))
    - often, just X0 ... Xj-1
      - siblings to left in parse tree
  - pass semantic information down the tree

Attribute Grammars (continued)

- Intrinsic attributes
  - synthesized attributes of leaves of parse tree
  - determined from outside tree
    - e.g., symbol table

Attribute Grammars (continued)

Example: expressions of the form id + id
- id's can be either int_type or real_type
- types of the two id's must be the same
- type of the expression must match its expected type

BNF: <expr> -> <var> + <var>
<var> -> id

Attributes:
actual_type - synthesized for <var> and <expr>
expected_type - inherited for <expr>
env - inherited for <expr> and <var>

Attribute Grammars (continued)

- Think of attributes as variables in the parse tree, whose values are calculated at compile time
  - conceptually, after parse tree is built
- Example attributes
  - actual_type
    - intrinsic for variables
    - determined from types of child nodes for <expr>
  - expected_type
    - for <expr>, determined by type of variable on LHS of assignment statement, for example
  - env
    - pointer to correct symbol table environment, to be sure semantic information used is correct set
      - think of different variable scopes

Attribute Grammars (continued)

Attribute Grammar:

1. syntax rule: <expr> -> <var>[1] + <var>[2]
   semantic rules:
     <var>[1].env <- <expr>.env
     <var>[2].env <- <expr>.env
     <expr>.actual_type <- <var>[1].actual_type
   predicate:
     <var>[1].actual_type = var>[2].actual_type
     <expr>.expected_type = <expr>.actual_type

2. syntax rule: <var> -> id
   semantic rule:
     <var>.actual_type <- lookup (id,<var>.env)

Computing Attribute Values

- If all attributes were inherited, could "decorate" the tree top-down
- If all attributes were synthesized, could decorate the tree bottom-up
- Usually, both kinds are used
  - use both top-down and bottom-up approaches
  - actual determination of order can be complicated, requiring calculations of dependency graphs
- One order that works for this simple grammar is on the next slide

Computing Attribute Values (continued)

1. <expr>.env <- inherited from parent
   <expr>.expected_type <- inherited from parent

2. <var>[1].env <- <expr>.env
   <var>[2].env <- <expr>.env

3. <var>[1].actual_type <- lookup(A,<var>[1].env)
   <var>[2].actual_type <- lookup (B,<var>[2].env)
   <var>[1].actual_type =? <var>[2].actual_type

4. <expr>.actual_type <- <var>[1].actual_type
   <expr>.actual_type =? <expr>.expected_type

Status of Attribute Grammars

- Well-defined, well-understood formalism
  - used for several practical compilers
- Grammars for real languages can become very large and cumbersome
  - and take significant amounts of computing time to evaluate
- Very valuable in a less formal way for actual compiler construction

Dynamic Semantics

- Describe the meaning of PL constructs
- No single widely accepted way of defining
- Three approaches used
  - operational semantics
  - axiomatic semantics
  - denotational semantics
- All are still in research stage, rather than practical use
  - most real compilers use ad-hoc methods

Operational Semantics

- Describe meaning of a program by executing its statements on a machine
  - actual or simulated
  - change of state of machine (values in memory, registers, etc.) defines meaning
- Could use actual hardware machine
  - too expensive
- Could use a software interpreter
  - too complicated, because of underlying machine complexity
  - not transportable

Operational Semantics (continued)

- Most common approach is to use simulator for simple, idealized (abstract) machine
  - build a translator (source code to machine code of simulated machine)
  - build a simulator
  - describe state transformations of simulated machine for each PL construct
- Evaluation
  - good if used informally
    - can have circular reasoning, since PL is being defined in terms of another PL
  - extremely complex if used formally
    - VDL description of semantics of PL/I was several hundred pages long

Axiomatic Semantics

- Define meaning of PL construct by effect on logical assertions about constraints on program variables
  - based on predicate calculus
  - approach comes from program verification
- Precondition is an assertion before a PL statement
  - states relationships and constraints among variables before statement is executed
- Postcondition is an assertion following a statement
  - {P} statement {Q}

Axiomatic Semantics (continued)

- Weakest precondition is least restrictive precondition that will guarantee postcondition
- a := b + 1 {a > 1}
  - possible precondition: {b > 10}
  - weakest precondition: {b > 0}

Axiomatic Semantics (continued)

- Axiom is a logical statement assumed to be true
- Inference rule is a method of inferring the truth of one assertion based on other true assertions
  - basic form for inference rule is
  - if S1, ..., Sn are true, S is true

$$\frac{S1, S2, \ldots, Sn}{S}$$

Axiomatic Semantics (continued)

- Then to prove a program
  - postcondition for program is desired result
  - work back through the program determining preconditions
    - which are postconditions for preceding statement
  - if precondition on first statement is same as program specification, program is correct
- To define semantics for a PL
  - define axiom or inference rule for each statement type in the language

Axiomatic Semantics Examples

An axiom for assignment statements:

$\{Q_{x->E}\}$ x := E {Q}

$Q_{x->E}$ means evaluate Q with E substituted for X

The Rule of Consequence:

{P} S {Q}, P' => P, Q => Q'

{P'} S {Q'}

Axiomatic Semantics Examples (continued)

An inference rule for sequences

- For a sequence:

```
{P1} S1 {P2}
{P2} S2 {P3}
```

the inference rule is:

```
{P1} S1 {P2}, {P2} S2 {P3}
---------------------------
      {P1} S1; S2 {P3}
```

Axiomatic Semantic Examples (continued)

An inference rule for logical pretest loops

For the loop construct:

{P} while B do S end {Q}

the inference rule is:

$$\frac{(I \text{ and } B)\ S\ \{I\}}{\{I\}\ \text{while B do S}\ \{I \text{ and (not B)}\}}$$

where I is the loop invariant.

Loop Invariant Characteristics

- The loop invariant I must meet the following conditions:
  - P => I
    - the loop invariant must be true initially
  - {I} B {I}
    - evaluation of the Boolean must not change the validity of I
  - {I and B} S {I}
    - I is not changed by executing the body of the loop
  - (I and (notB)) => Q
    - if I is true and B is false, Q is implied
  - The loop terminates
    - this can be difficult to prove

Axiomatic Semantics Evaluation

- Developing axioms or inference rules for all statements in a PL is difficult
  - Hoare and Wirth failed for function side effects and goto statements in Pascal
  - limiting a language to those statements that can have such rules written is too restrictive
- Good tool for research in program correctness and reasoning about programs
- Not practically useful (yet) for language designers and compiler writers

Denotational Semantics

- Define meaning by mapping PL elements onto mathematical objects whose behavior is rigorously defined
  - based on recursive function theory
  - most abstract of the dynamic semantics approaches
- To build a denotational specification for a language:
  - define a mathematical object for each language entity
  - define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

Denotational Semantics (continued)

- The meaning of language constructs are defined by only the values of the program's variables
  - in operational semantics the state changes are defined by coded algorithms
  - in denotational semantics, they are defined by rigorous mathematical functions
- The state of a program is the values of all its current variables
- Assume VARMAP is a function that, when given a variable name and a state, returns the current value of the variable
  - VARMAP(i_j, s) = v_j (the value of i_j in state s)

Denotational Semantics (continued)

- Consider some examples

Expressions

M_e(E, s): if VARMAP(i, s) = undef for some i in E
then error
else E′, where E′ is the result of
evaluating E after setting each
variable i in E to VARMAP(i, s)

Assignment Statements

M_a(x:=E, s): if M_e(E, s) = error
then error
else s′ = {<i_1′,v_1′>,...,<i_n′,v_n′>},
where for j = 1, 2, ..., n,
v_j′= VARMAP(i_j, s) if i_j <> x
= M_e(E, s) if i_j = x

Denotational Semantics (continued)

Logical Pretest Loops

M_l(while B do L, s) : if M_b(B, s) = undef
then error
else if M_b(B, s) = false
then s
else if M_{sl}(L, s) = error
then error
else M_l(while B do L,
M_{sl}(L, s))

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors
  - if the Boolean B is true, the meaning of the loop (state) is the meaning of the loop executed in the state caused by executing the loop body once
- In essence, the loop has been converted from iteration to recursion
  - recursion is easier to describe with mathematical rigor than iteration

Denotational Semantics Evaluation

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
  - complex descriptions imply complex language features
- Has been used in compiler generation systems
  - but not with practical effect
- Not useful as descriptive mechanism for language users