3b
Semantics

Semantics Overview

- Syntax is about *form* and semantics *meaning*
  - Boundary between syntax & semantics is not always clear
- First we'll motivate why semantics matters
- Then we'll look at issues close to the syntax end (e.g., *static semantics*) and *attribute grammars*
- Finally we'll sketch three approaches to defining "deeper" semantics:
  (1) Operational semantics
  (2) Axiomatic semantics
  (3) Denotational semantics

Motivation

- Capturing what a program in some programming language **means** is very difficult
- We can't really do it in any practical sense
  - For most work-a-day programming languages (e.g., C, C++, Java, Perl, C#, Python)
  - For large programs
- So, why is worth trying?

Motivation: Some Reasons

- How to convey to the programming language compiler/interpreter writer what she should do?
  - Natural language may be too ambiguous
- How to know that the compiler/interpreter did the *right thing* when it executed our code?
  - We can't answer this w/o a very solid idea of what the *right thing* is
- How to be sure that the program satisfies its specification?
  - Maybe we can do this automatically if we know what the program means

Program Verification

- **[Program verification](#)** is the process of formally proving that the computer program does exactly what is stated in the program's specification
- Program verification can be done for simple programming languages and small or moderately sized programs
- It requires a *formal specification* for what the program should do – e.g., its inputs and the actions to take or output to generate given the inputs
- That's a hard task in itself!

Program Verification

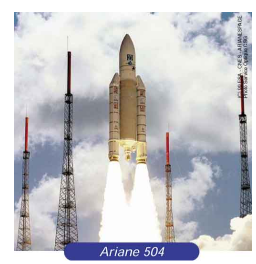
- There are applications where it is worth it to
  - (1) use a simplified programming language
  - (2) work out formal specs for a program
  - (3) capture the semantics of the simplified PL and
  - (4) do the hard work of putting it all together and proving program correctness.
- What are they?

Program Verification

- There are applications where it is worth it to (1) use a simplified programming language, (2) work out formal specs for a program, (3) capture the semantics of the simplified PL and (4) do the hard work of putting it all together and proving program correctness. Like…
- Security and encryption
- Financial transactions
- Applications on which lives depend (e.g., healthcare, aviation)
- Expensive, one-shot, un-repairable applications (e.g., Martian rover)
- Hardware design (e.g. Pentium chip)

Double Int kills Ariane 5

- It took the European Space Agency 10 years and $7B to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe supremacy in the commercial space business



- All it took to explode the rocket less than a minute into its maiden voyage in 1996 was a small computer program trying to stuff a 64-bit number into a 16-bit space.

Intel Pentium Bug

- In the mid 90's a bug was found in the floating point hardware in Intel's latest Pentium microprocessor
- Unfortunately, the bug was only found after many had been made and sold
- The bug was subtle, effecting only the ninth decimal place of some computations
- But users cared
- Intel had to recall the chips, taking a $500M write-off

So…

- While automatic program verification is a long range goal …
- Which might be restricted to applications where the extra cost is justified
- We should try to design programming languages that help, rather than hinder, our ability to make progress in this area.
- We should continue research on the semantics of programming languages …
- And the ability to prove program correctness

Semantics

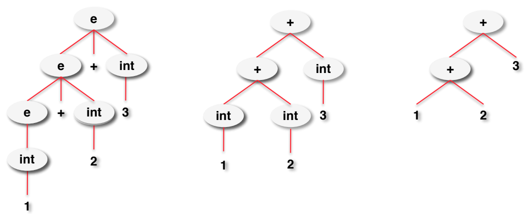
- Next we'll look at issues close to the syntax end, what some calls *static semantics*, and the technique of *attribute grammars*.
- Then we'll sketch three approaches to defining "deeper" semantics
  (1) Operational semantics
  (2) Axiomatic semantics
  (3) Denotational semantics

Static Semantics

- Static semantics covers some language features difficult or impossible to handle in a BNF/CFG
- It's also a mechanism for building a parser that produces an abstract syntax tree of its input
- Attribute grammars are one common technique
- Categories attribute grammars can handle:
  - Context-free but cumbersome (e.g., type checking)
  - Non-context-free (e.g., variables must be declared before they are used)

Parse tree vs. abstract syntax tree

- Parse trees follow a grammar and usually have many nodes that are artifacts of how the grammar was written
- An abstract syntax tree (AST) eliminates useless structural nodes
- It uses nodes corresponding to constructs in the programming language and is easier to interpret or generate code from it
- Consider 1 + 2 + 3:



parse tree   an AST   another AST

Attribute Grammars

- Attribute Grammars (AGs) were developed by Donald Knuth in ~1968
- Motivation:
  - CFGs can't describe all of the syntax of programming languages
  - Additions to CFGs to annotate the parse tree with some "semantic" info
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammar Example

- Ada has this rule to describe procedure definitions:
  <proc> => procedure <prName> <prBody> end <prName> ;
- The name after *procedure* must be the same as the name after *end*
- This is not possible to capture in a CFG (in practice) because there are too many names
- Solution: annotate parse tree nodes with attributes and add a "semantic" rules or constraints to the syntactic rule in the grammar

rule: <proc> => procedure <prName>[1] <prBody> end <prName>[2] ;
constraint: <prName>[1].string == <prName>[2].string

Attribute Grammars

Def: An *attribute grammar* is
G=(S,N,T,P)

A Grammar is formally defined by specifying four components.
- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols
- P is a set of productions or rules

with the following additions:
- For each grammar symbol *x* there is a set A(*x*) of attribute values
- Each rule has a set of functions that define certain attributes of the non-terminals in the rule
- Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars

- Let $X_0 \Rightarrow X_1 \ldots X_n$ be a grammar rule
- Functions of the form $S(X_0) = f(A(X_1),\ldots A(X_n))$ define *synthesized attributes*
  - i.e., attribute defined by a nodes children
- Functions of the form $I(X_j) = f(A(X_0),\ldots A(X_n))$ for $i \le j \le n$ define *inherited attributes*
  - i.e., attribute defined by parent and siblings
- Initially, there are *intrinsic attributes* on the leaves
  - i.e., attribute predefined

Attribute Grammars

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>`

CMSC 331, Some material © 1998 by Addison Wesley Longman, Inc.

Attribute Grammars

Attribute Grammar:

1. Syntax rule: <expr> -> <var>[1] + <var>[2]
   Semantic rules:
   <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_type (id, <var>)

Compilers usually maintain a "symbol table" where they record the names of procedures and variables along with type type information. Looking up this information in the symbol table is a common operation.

CMSC 331, Some material © 1998 by Addison Wesley Longman, Inc.

Attribute Grammars (continued)

How are attribute values computed?

- If all attributes were inherited, the tree could be *decorated* in top-down order
- If all attributes were synthesized, the tree could be *decorated* in bottom-up order
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used

CMSC 331, Some material © 1998 by Addison Wesley Longman, Inc.

Attribute Grammars (continued)

Suppose we process the expression A+B
using rule <expr> -> <var>[1] + <var>[2]

<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A, <var>[1])
<var>[2].actual_type ← lookup (B, <var>[2])
<var>[1].actual_type == <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type == <expr>.expected_type

Attribute Grammar Summary

- AGs are a practical extension to CFGs that allow us to annotate the parse tree with information needed for semantic processing
  - e.g., interpretation or compilation
- We call the annotated tree an *abstract syntax tree*
  - It no longer just reflects the derivation
- AGs can move information from anywhere in abstract syntax tree to anywhere else in a controlled way
  - Needed for no-local syntactic dependencies (e.g., Ada example) and for semantics

Static vs. Dynamic Semantics

- Attribute grammar is an example of *static semantics* (e.g., type checking) that don't reason about how things *change* when a program is executed
- Understanding what a program means often requires reasoning about how, for example, a variable's value changes
- *Dynamic semantics* tries to capture this
  - E.g., proving that an array index will never be out of its intended range

Dynamic Semantics

- No single widely acceptable notation or formalism for describing dynamic semantics
- Here are three approaches we'll briefly examine:
  - Operational semantics
  - Axiomatic semantics
  - Denotational semantics

Dynamic Semantics

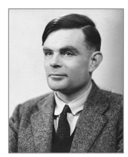
- **Q:** How might we define what expression in a language mean?
- **A:** One approach is to give a general mechanism to *translate* a sentence in L into a set of sentences in another language or system that is well defined
- For example:
  - Define the meaning of computer science terms by translating them in ordinary English
  - Define the meaning of English by showing how to translate into French
  - Define the meaning of French expression by translating into mathematical logic

[turtles all the way down]

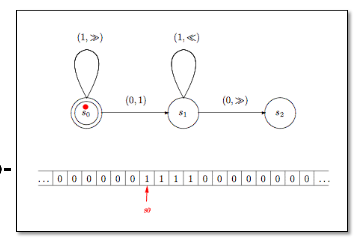
Operational Semantics

- Idea: describe the meaning of a program in language L by specifying how statements effect the *state of a machine* (simulated or actual) when executed.
- The change in the state of the machine (memory, registers, stack, heap, etc.) defines the meaning of the statement
- Similar in spirit to the notion of a *Turing Machine* and also used informally to explain higher-level constructs in terms of simpler ones.

Alan Turing and his Machine



- The Turing machine is an *abstract machine* introduced in 1936 by Alan Turing
  - – Alan Turing (1912 –54) was a British mathematician, logician, cryptographer, considered a father of modern computer science
- It can be used to give a mathematically precise definition of algorithm or 'mechanical procedure'
- Concept widely used in theoretical computer science, especially in complexity theory and the theory of computation



Operational Semantics

- This is a common technique
- Here's how we might explain the meaning of the for statement in C in terms of a simpler reference language:

c statement	operational semantics

`for(e1;e2;e3) {<body>}`	`e1;`
	`loop: if e2=0 goto exit`
	`<body>`
	`e3;`
	`goto loop`
	`exit:`

Operational Semantics

- To use operational semantics for a high-level language, a virtual machine in needed
- A *hardware* pure interpreter is too expensive
- A *software* pure interpreter also has problems:
  - The detailed characteristics of the particular computer makes actions hard to understand
  - Such a semantic definition would be machine-dependent

Operational Semantics

A better alternative: a complete computer simulation

- Build a translator (translates source code to the machine code of an idealized computer)
- Build a simulator for the idealized computer

Evaluation of operational semantics:

- Good if used informally
- Extremely complex if used formally (e.g. VDL)

Vienna Definition Language

- VDL was a language developed at IBM Vienna Labs as a language for formal, algebraic definition via operational semantics.
- It was used to specify the semantics of PL/I
- See: *The Vienna Definition Language*, P. Wegner, ACM Comp Surveys 4(1):5-63 (Mar 1972)
- The VDL specification of PL/I was very large, very complicated, a remarkable technical accomplishment, and of little practical use.

The Lambda

What's a calculus, anyway?
"A method of computation or calculation in a special notation (as of logic or symbolic logic)" --
Merriam-Webster

- The first use of operati the *lambda calculus*
  - A formal system designed to investigate function definition, function application and recursion
  - Introduced by Alonzo Church and Stephen Kleene in the 1930s
- The lambda calculus can be called the smallest universal programming language
- It's widely used today as a target for defining the semantics of a programming language

The Lambda Calculus

- The lambda calculus consists of a single transformation rule (variable substitution) and a single function definition scheme
- The lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism
- We'll revisit the lambda calculus later in the course
- The Lisp language is close to the lambda calculus model

The Lambda Calculus

- The lambda calculus
  - –introduces variables ranging over values
  - –defines functions by (lambda) abstracting over variables
  - –applies functions to values
- Examples:

  simple expression: $x + 1$

  function that adds one to its arg: $\lambda x.\ x + 1$

  applying it to 2: $(\lambda x.\ x + 1)\ 2$

Operational Semantics Summary

- The basic idea is to define a language's semantics in terms of a reference language, system or machine
- It's use ranges from the theoretical (e.g., lambda calculus) to the practical (e.g., Java Virtual Machine)

Axiomatic Semantics

- Based on formal logic (first order predicate calculus)
- *Original purpose:* formal program verification
- *Approach:* Define axioms and inference rules in logic for each statement type in the language (to allow transformations of expressions to other expressions)
- The expressions are called *assertions* and are either
  - **Preconditions:** An assertion before a statement states the relationships and constraints among variables that are true at that point in execution
  - **Postconditions:** An assertion following a statement

Logic 101

Propositional logic:

Logical constants: true, false

Propositional symbols: P, Q, S, ... that are either true or false

Logical connectives: $\wedge$ (and) , $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (is equivalent), $\neg$ (not) which are defined by the truth tables below.

Sentences are formed by combining propositional symbols, connectives and parentheses and are either true or false. e.g.: $P \wedge Q \Leftrightarrow \neg(\neg P \vee \neg Q)$

First order logic adds

(1) Variables which can range over objects in the domain of discourse

(2) Quantifiers including: $\forall$ (forall) and $\exists$ (there exists)

(3) Predicates to capture domain classes and relations

Examples: $(\forall p)(\forall q)\ p \wedge q \Leftrightarrow \neg(\neg p \vee \neg q)$

$\forall x\ prime(x) \Rightarrow \exists y\ prime(y) \wedge y>x$

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

LOGIC, LIKE WHISKY

loses its beneficial effects
when taken in large quantities

Lord Dunsany

Axiomatic Semantics

- Axiomatic semantics is based on Hoare Logic (after computer scientists Sir Tony Hoare)
- Based on *triples* that describe how execution of a statement changes the state of the computation
- Example: {P} S {Q} where
  - P is a logical statement of what's true before executing S
  - Q is a logical expression describing what's true after
- In general we can reason forward or backward
  - Given P and S determine Q
  - Given S and Q determine P
- Concrete example: {x>0} x = x+1 {x>1}

Axiomatic Semantics

A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

Notation:

{P} Statement {Q}

precondition **postcondition**

Example:

{?} a := b + 1 {a > 1}

We often need to infer what the precondition must be for a given post-condition

One possible precondition: {b>10}

Another: {b>1}

Weakest precondition: {b > 0}

Weakest Precondition?

- A *weakest precondition* is the least restrictive precondition that will guarantee the post-condition
- There are an *infinite* number of possible preconditions P? that satisfy

  {P?} a := b + 1 {a > 1}
- Namely b>0, b>1, b>2, b>3, b>4, …
- The weakest precondition is one that *logically is implied by* all of the (other) preconditions
  - b>1 => b>0
  - b>2 => b>0
  - b>3 => b>0
  - …

Axiomatic Semantics in Use

Program proof process:

- The post-condition for the whole program is the desired results
- Work back through the program to the first statement
- If the precondition on the first statement is the same as (or implied by) the program specification, the program is correct

Example: Assignment Statements

Here's how we might define a simple assignment statement of the form ***x := e*** in a programming language.

- $\{Q_{x\text{->}E}\}$ x := E {Q}
- Where $Q_{x\text{->}E}$ means the result of replacing all occurrences of *x* with *E* in *Q*

So from

{Q} a := b/2-1 {a<10}

We can infer that the weakest precondition Q is b/2-1<1 which can be rewritten as or b<22

Axiomatic Semantics

- *The Rule of Consequence:*

$$\frac{\{P\}\ S\ \{Q\},\ \ P' \Rightarrow P,\ \ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

- *An inference rule for sequences*

for a sequence S1 ; S2:

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

the inference rule is:

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

A notation from symbolic logic for specifying a rule of inference with premise P and consequence Q is

$$\frac{P}{Q}$$

e.g., modus ponens can be specified as:

$$\frac{P,\ P \Rightarrow Q}{Q}$$

Conditions

Here's a rule for a conditional statement

$$\frac{\{B \wedge P\}\ S1\ \{Q\},\ \{\neg B \wedge P\}\ S2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{Q\}}$$

And an example of its use for the statement

{P} if x>0 then y=y-1 else y=y+1 {y>0}

So the weakest precondition P can be deduced as follows:

The postcondition of S1 and S2 is Q.

The weakest precondition of S1 is $x>0 \wedge y>1$ and for S2 is $x<=0 \wedge y>-1$

The rule of consequence and the fact that $y>1 \Rightarrow y>-1$ supports the conclusion

That the weakest precondition for the entire conditional is $y>1$.

Conditional Example

Suppose we have:

{P}

If x>0 then y=y-1 else y=y+1

{y>0}

Our rule

$$\frac{\{B \wedge P\}\ S1\ \{Q\},\ \{\neg B \wedge P\}\ S2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{Q\}}$$

Consider the two cases:

– x>0 and y>1

– x<=0 and y>-1

- What is a (weakest) condition that implies both y>1 and y>-1

Conditional Example

- What is a (weakest) condition that implies both y>1 and y>-1?
- Well y>1 implies y>-1
- y>1 is the weakest condition that ensures that after the conditional is executed, y>0 will be true.
- Our answer then is this:

{y>1}

If x>0 then y=y-1 else y=y+1

{y>0}

Loops

For the loop construct {P} while B do S end {Q} the inference rule is:

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S\ \{I \wedge \neg B\}}$$

where I is the *loop invariant*, a proposition necessarily true throughout the loop's execution

- I is true before the loop executes and also after the loop executes
- B is false after the loop executes

Loop Invariants

A loop invariant *I* must meet the following conditions:

1. P => I (the loop invariant must be true initially)
2. {I} B {I} (evaluation of the Boolean must not change the validity of I)
3. {I and B} S {I} (I is not changed by executing the body of the loop)
4. (I and (not B)) => Q (if I is true and B is false, Q is implied)
5. The loop terminates (this can be difficult to prove)

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.
- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs
- It is much less useful for language users and compiler writers

Denotational Semantics

- A technique for describing the meaning of programs in terms of mathematical functions on programs and program components.
- Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory.
- Originally developed by Scott and Strachey (1970) and based on recursive function theory
- The most abstract semantics description method

Denotational Semantics

- The process of building a denotational specification for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables

Denotational Semantics (continued)

The difference between denotational and operational semantics: 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

$$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable

$$\text{VARMAP}(i_j, s) = v_j$$

Example: Decimal Numbers

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0|1|2|3|4|5|6|7|8|9)

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9
M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)
M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1
…
M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Expressions

```
M_e(<expr>, s) Δ=
  case <expr> of
    <dec_num> => M_dec(<dec_num>, s)
    <var> =>
        if VARMAP(<var>, s) = undef
            then error
            else VARMAP(<var>, s)
    <binary_expr> =>
        if (M_e(<binary_expr>.<left_expr>, s) = undef
            OR M_e(<binary_expr>.<right_expr>, s) =
                        undef)
            then error
            else
              if (<binary_expr>.<operator> = '+' then
                M_e(<binary_expr>.<left_expr>, s) +
                     M_e(<binary_expr>.<right_expr>, s)
              else M_e(<binary_expr>.<left_expr>, s) *
                M_e(<binary_expr>.<right_expr>, s)
```


Assignment Statements

M_a(x := E, s) Δ=
if M_e(E, s) = error
then error
else s' = $\{<i_1',v_1'>,<i_2',v_2'>,\ldots,<i_n',v_n'>\}$,
where for j = 1, 2, ..., n,
v_j' = VARMAP(i_j, s) if i_j <> x
= M_e(E, s) if i_j = x

Logical Pretest Loops

M_l(while B do L, s) Δ=

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))

Logical Pretest Loops

- 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
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
- Recursion, when compared to iteration, is easier to describe with mathematical rigor

Denotational Semantics

Evaluation of denotational semantics:

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems

Summary

This lecture we covered the following

- Backus-Naur Form and Context Free Grammars
- Syntax Graphs and Attribute Grammars
- Semantic Descriptions: Operational, Axiomatic and Denotational