Program Semantics

- **Semantics:** what is the “meaning” of a program?
  - What is its run-time behavior?
  - What does the program actually do?

- Originally, these questions had an easy answer:
  - Run it and find out!

- A disadvantage, of course, was that a program could then mean (i.e., do) different things on different machines

Making Semantics Precise

- By the 1960’s, researchers realized that advancing programming meant advancing the art of doing semantics

- If there is **no** fixed thing a program **must** do when run:
  - How do we solve complex problems?
  - How do we write compilers for a given language?
  - What does “de-bugging” even mean?

- A variety of approaches tried to answer questions like:
  - What behavior should be **expected** from a program?
  - How can we **determine** what programs will (likely) do?
  - Can we **prove** that a program will do one thing or another?
An Empirical Approach: Operational Semantics

- Translate program into a precise series of steps, as they would be executed by a specific machine
- A combination of some well-known real architecture and compiler; or…
- A particular virtual machine, or…
- A defined abstract machine simulation
- Using that specification, we can outline how the program will work
  - For virtual/abstract machines, can still be question of how they work…

A Formal Approach: Attribute Grammars

- Developed by Donald Knuth and Peter Wegners (1960’s)
  - A grammar that mirrors the CFG for a language
  - Popular with those who study functional languages
  - For each production in a CFG, the attribute grammar provides a matching semantic rule
  - The rule will use semantic functions to give the meaning of the productions where necessary
  - These functions are designed/written by the language designers and/or compiler writers, perhaps using another programming language, or well-understood mathematical conventions

An Example Attribute Grammar

- For the CFG on the left, we have attribute grammar on the right
  - The symbols of the CFG here are $E,T,F$
  - The subscripts only have meaning for the semantics, not the syntax, allowing us to disambiguate uses of the same non-terminal on the left/right of a production

Copy Rules

- The most simple rules in the attribute grammar simply copy over a value, from an expression on the right of a production to one on the left
- The values of constants (const.val) is determined by basic language rules for various literal types (integers here)
Applying Semantic Functions

1. \( E_1 \rightarrow E_2 + T \)  \( \triangleright \ E_2, Val := \text{sum}(E_2, Val) \)
2. \( E_1 \rightarrow E_2 - T \)  \( \triangleright \ E_2, Val := \text{difference}(E_2, Val) \)
3. \( E \rightarrow T \)  \( \triangleright \ E, Val := T, Val \)
4. \( T_1 \rightarrow T_2 * F \)  \( \triangleright \ T_1, Val := \text{product}(T_1, Val) \)
5. \( T_1 \rightarrow T_2 / F \)  \( \triangleright \ T_1, Val := \text{quotient}(T_2, Val, F, Val) \)
6. \( T \rightarrow F \)  \( \triangleright \ T, Val := \text{F, Val} \)
7. \( F_1 \rightarrow - F \)  \( \triangleright \ F_1, Val := \text{additive, inverse}(F, Val) \)
8. \( F \rightarrow ( E ) \)  \( \triangleright \ F, Val := E, Val \)
9. \( F \rightarrow \text{const} \)  \( \triangleright \ F, Val := \text{const, Val} \)

- The other rules apply semantic functions to values
- Here, the functions sum, difference, product, etc. are defined in their usual mathematical ways

An SLR(1) Grammar

1. \( E_1 \rightarrow E_2 + T \)  \( \triangleright \ E_2, Val := \text{sum}(E_2, Val) \)
2. \( E_1 \rightarrow E_2 - T \)  \( \triangleright \ E_2, Val := \text{difference}(E_2, Val) \)
3. \( E \rightarrow T \)  \( \triangleright \ E, Val := T, Val \)
4. \( T_1 \rightarrow T_2 * F \)  \( \triangleright \ T_1, Val := \text{product}(T_2, Val, F, Val) \)
5. \( T_1 \rightarrow T_2 / F \)  \( \triangleright \ T_1, Val := \text{quotient}(T_2, Val, F, Val) \)
6. \( T \rightarrow F \)  \( \triangleright \ T, Val := \text{F, Val} \)
7. \( F_1 \rightarrow - F \)  \( \triangleright \ F_1, Val := \text{additive, inverse}(F, Val) \)
8. \( F \rightarrow ( E ) \)  \( \triangleright \ F, Val := E, Val \)
9. \( F \rightarrow \text{const} \)  \( \triangleright \ F, Val := \text{const, Val} \)

- This is the simplest sort of situation, using only synthetic attributes: each symbol gets a value assigned to it only for rules in which it appears on the left-hand side of a rule
- The original syntax is LR(1), and the semantics is thus SLR(1)

Another Formal Approach: Denotational Semantics

- Christopher Strachey & Dana Scott (1970's)
- Based on the idea of state transformations
- Treats a program as a set of functions that manipulate its environment and state:
  - Environment: set of objects/types that are active
  - State: the set of all active objects and their values
Not to Be Confused with: Detonational Semantics

- A more controversial approach that was soon abandoned
  - Dealt with programs whose operations caused computers to explode
  - Today and in future, access to increasingly affordable hardware means that this may be an area for interesting future work

Denotational Semantics

- Consists of three parts:
  1. Abstract syntax of the language
  2. Semantic algebra defining a computational model
     - Mapping of things to types like integers, booleans, etc.
     - State (a set of associations)
     - Operations on those types of things
  3. Mapping functions that translate expressions in the language to the semantic algebra

- Programs map to sequences of state transformations
  - The meaning of a program is the final state in this sequence

Semantic Domains

- A semantic domain is any set whose properties and operations are well understood
  - Program operations are functions that take inputs from some domain-set and produce outputs from the same or other domain

  Examples include:
  - Integers
  - Rational numbers
  - Boolean values
  - Strings
  - …

Domain Mappings

- Let \( \gamma \) be a mapping from variables to memory locations
- Let \( \mu \) be a mapping from memory locations to values
  \[
  \begin{align*}
  i &= 13; \\
  j &= -1;
  \end{align*}
  \]

- We can represent these mappings formally to say, e.g., that the assignments above produce the mappings:
  \[
  \begin{align*}
  \gamma(i) &= 154 \\
  \gamma(j) &= 155 \\
  \mu(154) &= 13 \\
  \mu(155) &= -1
  \end{align*}
  \]

That is, variable \( i \) corresponds to memory location 154, which stores the value 13, and variable \( j \) corresponds to memory location 154, which stores the value \(-1\).
For the sake of simplicity, we can represent such formal functional mappings as sets consisting of all active <variable, address> or <address, value> pairs.

"undef" means that no value has been stored in a given memory location by the program:

\[
\begin{align*}
\gamma &= \{<i, 154>, <j, 155>\} \\
\mu &= \{0, \text{undef}, \ldots, <154, 13>, <155, -1>, \ldots\}
\end{align*}
\]

Domain Mappings

γ(i) = 154
γ(j) = 155
μ(154) = 13
μ(155) = -1

For the sake of simplicity, we can represent such formal functional mappings as sets consisting of all active <variable, address> or <address, value> pairs.

Program State

- The state, σ, of a program is a concatenation of the other two mappings.
- For our example code, we can define the state of variables i and j as:
  \[
  \sigma(i) = \mu(\gamma(i)) = \mu(154) = 13 \\
  \sigma(j) = \mu(\gamma(j)) = \mu(155) = -1
  \]
- To simplify further, we will ignore low-level memory addressing and treat state as a function from variables directly to values (again writing as a simple set):

\[
\sigma = \{<i, 13>, <j, -1>\}
\]

State Functions

- In general, we will define the function \( \sigma(exp) \) to be a function that returns the value of the expression, \( exp \), given state \( \sigma \)

\[
\sigma = \{<i, 13>, <j, -1>\}
\]

\[
\begin{align*}
\sigma(i) &= 13 \\
\sigma(i + j) &= 12
\end{align*}
\]

State Transformations

- Program statements usually change the state

\[
\begin{align*}
\sigma_0 &= \{<x, 1>, <y, 2>, <z, 3>\} \\
y &= 2 * z + 3; \\
\sigma_1 &= \{<x, 1>, <y, 9>, <z, 3>\} \\
w &= 4; \\
\sigma_2 &= \{<x, 1>, <y, 9>, <z, 3>, <w, 4>\}
\end{align*}
\]
Program Meaning
- Let $\Sigma$ be the set of all program states, $\sigma$
- We let $M$ be a meaning function
  - **Inputs:** one or more syntactic symbols, current program state
  - **Output:** new program state (changed or not)
- For a language with set of symbols (terminal or non-terminal), $S$, the meaning function is therefore a mapping:
  $$ M : (S^k \times \Sigma) \rightarrow \Sigma \quad (k \geq 1) $$

Meaning Functions
- Meaning functions are **partial functions**
  - That is, they may not be defined on every possible input
  - Like Java's `Math.sqrt(x)`, which returns `NaN` if $x < 0$
- There may be some expressions that are syntactically valid, but which do not have well defined meaning:
  ```
  for (i = 1; i >= 0; i++) i--;
  ```
  - No definite final state, since it loops forever, alternating between ($i == 0$) and ($i == 1$)
  - In a denotational sense, this code has no real meaning

Program Meaning
- Consider a simple programming language with integer variables that can be declared and assigned:
  ```
  Program --> Declaration* Main
  Main --> main() { Declaration* Assignment* }
  Declaration --> int Variable;
  Assignment --> Variable = Int;
  Variable --> Letter Letter*;
  Letter --> a | b | ... | z
  Int --> 0 | LeadDigit Digit*;
  LeadDigit --> 1 | 2 | ... | 9
  Digit --> 0 | 1 | ... | 9
  ```
  - Assuming our parser ignores white-space between symbols, a legal program might be:
  ```
  int i; int j;
  main() {
      int mine;
      i = 13;
      j = 1;
      mine = 0;
  }
  ```
- The meaning function gives us rules for interpreting a program
  - The meaning of a program is given by the value of that mapping, using an initially **empty state**:
    $$ M(Program, \emptyset) = M(Declaration^* Main, \emptyset) $$
  - This is in turn composed of the meanings of its two parts, according to the semantic rule:
    $$ M(Declaration^* Main, \emptyset) = M(Main, M(Declaration^*, \emptyset)) $$
Program Meaning

\[ M(\text{Program}, 0) = M(\text{Declaration}^* \text{ Main}, 0) \]
\[ M(\text{Declaration}^* \text{ Main}, 0) = M(\text{Main}, M(\text{Declaration}^*, 0)) \]

- The meaning of a (possibly empty) block of variable declarations is the current state, \( \sigma_i \), with all the declared variables—each initially with undefined value—added to it (we will assume that redefining variables is never allowed):

\[
M(\text{Declaration}^*, \sigma_i) = \begin{cases} 
\sigma_i & \text{if } \epsilon \\
\sigma_i \cup \{(v_1, \text{undefined}), (v_2, \text{undefined}), \ldots, (v_k, \text{undefined})\}, & \text{else}
\end{cases}
\]

- For a block of declarations that precedes our `main()` function, this will be the initial state of the program, \( \sigma_0 \):

\[
M(\text{Declaration}^* \text{ Main}, 0) = M(\text{Main}, M(\text{Declaration}^*, 0))
\]

\[ = M(\text{Main}, \sigma_0) \]

This Week & Next

- **Topic:** Semantics
- **Reading:** Text, 4.1–4.3
- **Homework 02:** Friday, 08 March, 5:00 PM on D2L
- **Midterm Exam:** Friday, 15 March, in class
  - Open book, open notes
  - Covers 'formal' material to denotational semantics (not Scala programming)
  - Practice exam available by this weekend
- **Office Hours:** Wing 210
  - Monday, 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - No office hours Thursday
  - Friday: 9:00 AM – 10:30 AM