Forth: Details
Specifying Semantics

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, March 24, 2017

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

© 2017 Glenn G. Chappell
We discussed file I/O in Forth.

See io.fs.
Forth: Details
Stacks

A Forth implementation adhering to the ANSI standard is actually required to have four stacks.

Data stack
Holds integer values. These are also used as pointers and booleans. This is the stack we have been dealing with.

Floating-point stack
Holds floating-point values.

Return stack
Holds return addresses for words that are called.

Locals stack
Holds local variables.

Why are the return stack & locals stack separate? I do not know.
Forth: Details
Floating Point [1/3]

Forth includes support for floating-point computations. Floating-point values (essentially the same as C/C++ values of type `double`) are stored on a separate stack: the floating-point stack.

Floating-point literals must contain “e” or “E”. These push a value on the floating-point stack.

\[
-4e \\
1.2E \\
1.2e17
\]
Words that handle floating-point are often named the same as the corresponding integer-handling words, with an “f” prepended.

```
f.  \ Like .
f.s \ Like .s
fdup  ( F: x -- x x ) \ Like dup Also fdrop fswap ... 
f+  ( F: x y -- x+y ) \ Like + Also f- f* f/
```

Here are some other floating-point-handling words.

```
f**  ( F: x y -- pow[x,y] ) \ x raised to the y power
fsqrt ( F: x -- sqrt[x] )
fexp  ( F: x -- exp[x] ) \ Also flog fsin fcos ...
1/f  ( F: x -- 1/x )
```
TO DO

- Look at some Forth code that does floating-point computation.

Done. See float.fs.
Forth: Details
Other Features

Some Forth features that we do not have time to cover:

- Exceptions
  - Forth has a notion of exception that can be used for error handling.
- Defining new flow-of-control words
  - Some of the words we have covered are special: `if` `else` `endif` `begin` `while` `repeat` `?do` `loop` `recurse`. These affect the flow of control in ways that we do now know how to duplicate. But Forth does allow us to write such words ourselves.
- Defining new **defining words**
  - Some other words are special in another way: `variable` `constant` `:` `;`. These allow new words to be defined. Forth allows us to write this kind of word as well.
Now we take a brief look at semantics and how it is specified.

Recall:

- **Syntax** = structure (of code)
- **Semantics** = meaning (of code)

Grammatical Notes

- “Semantics” is an uncountable noun (like “butter”). It is mostly used in the singular (so “Semantics is ...”, not “Semantics are ...”).
- “Semantic” is the corresponding adjective.
How is semantics used?

- A **programmer** needs to know the semantics of a PL in order to write correct code.
- A design of a **compiler** needs to be based on the semantics of the source PL, so that correct object code can be generated.
- Similarly, the design of an **interpreter** needs to be based on the semantics of the source PL, so that correct actions can be performed.
- Semantics is useful in **optimization**: altering code so as to improve performance, while keeping semantics the same.
- Semantics is used in **verification**: checking that code performs the actions it is supposed to.
Semantics is generally divided into two kinds: static and dynamic.

**Static semantics** includes the aspects of semantics that can be checked before a program executes. This includes:

- Typing, in statically typed PLs.
- Dependencies (what relies on what).
- Other things like whether all cases in a `switch` are distinct.

**Dynamic semantics** refers to the semantics of a running program: what statements do, and what expressions compute. In a dynamically typed PL, this also includes typing.
We have looked at methods for formally specifying syntax—in particular, phrase-structure grammars. **Formal semantics** refers to methods for formally specifying semantics. These generally involve mathematical notations.

We look briefly at four formal-semantics methods. We will not cover notation.

- **Attribute grammars.** Specify static semantics via attributes added to AST nodes.
- **Operational semantics.** Specify dynamic semantics of a PL in terms of the semantics of some other PL or abstract machine (usually the latter).
- **Axiomatic semantics.** Specify dynamic semantics in terms logical statements about program state.
- **Denotational semantics.** Specify dynamic semantics by representing state & values with mathematical objects, commands & computations by functions.
Attribute grammars form a method for specifying static semantics. They involve adding information to the nodes of an AST and checking whether this information has certain required properties.

- **Attributes** are tags that can be added to the nodes of an AST.
- Some attributes are **synthesized**: they are determined based on the entity a node represents, along with attributes of child nodes.
- Other attributes are **inherited**: they are determined based on the attributes of parents and/or sibling nodes.
- There are **predicates** that tell whether attributes are semantically correct.

A typical attribute would be the type of the entity that a node represents.
Operational semantics is the name given to a number of methods for specifying dynamic semantics. These focus on the actions or computations that various pieces of code perform.

There are a number of different kinds of operational semantics. What they have in common is that the actions/computations of code are expressed in terms of some other system whose semantics is already known. This system might be:

- A mathematical formal system.
- An abstract machine—perhaps some kinds of automaton.
- Some other programming language.

For operational semantics to have any value, this other system must generally be, first, precisely specified and, second, simpler than the PL whose semantics is being described.
Axiomatic semantics is a method for specifying dynamic semantics. It specifies program operation in terms of logical statements about program state.

An invariant is a statement that is true at a particular point in a program.

A precondition is an invariant just before a section of code. It expresses what must be true for the code to be executed.

A postcondition is an invariant just after a section of code. It expresses what the code does, in terms of statements about state.

```
// Precondition: b != 0
c = a/b;
// Postcondition: c*b == a
```

Written in some hypothetical programming language: almost-but-not-quite C++.
Given a statement and a postcondition, axiomatic semantics allows us to determine the **minimal precondition** for the statement, that is, the least restrictive condition we can place on the state of the program before the statement, which will result in the postcondition being true after the statement.

// Minimal precondition: $b > 5$

```plaintext
a = b;
```

Find the minimal precondition based on the given postcondition.

// Given postcondition: $a > 5$

Other preconditions are possible here: $b \geq 28$, $(b == 173 && a < 0)$, etc. But $b > 5$ is the unique minimal precondition.
The primary use of axiomatic semantics is in **program verification**: proving that a program does what we want it to. Here is how this is done.

- Express the desired action of the program in terms of a postcondition for the entire program.
- Work backwards through the program, finding minimal preconditions. The minimal precondition for a statement then becomes the given postcondition for the previous statement.
- Finish by finding the minimal precondition for the entire program. This gives the conditions under which the program performs correctly.
- If there are *no* preconditions for the program, then we know that the program *always* performs correctly, and we have successfully verified it.
Denotational semantics is a method for specifying dynamic semantics. It involves the construction of mathematical objects called denotations, which describe the meanings of program entities. Denotational semantics is compositional: the denotation of an entity is described in terms of the denotations of the entities it is composed of.

Here is a ridiculously simple example.

Consider the following grammar.

\[
\begin{align*}
\text{digit} & \rightarrow '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \\
\text{number} & \rightarrow \text{digit} \\
\text{number} & \rightarrow \text{number} \text{ digit}
\end{align*}
\]
We describe a function $m$ that, given a program entity, returns its denotation. In this case, the denotation will be an integer.

- In the first production, for a digit $d$, $m(d)$ is the usual numeric value. So $m('0') = 0$, $m('1') = 1$, etc.
- In the second production, if a number $n$ is simply the digit $d$, then $m(n) = m(d)$.
- In the third production, if a number $n$ expands to a number $n_0$ followed by a digit $d$, then $m(n) = 10 \times m(n_0) + m(d)$. 
Formal methods for specifying *syntax* have been very successful. Virtually all PLs have a syntax specification in terms of a grammar.

However, formal semantics has been much less successful.

In practice, there are two ways the semantics of a PL might be specified.

- Entirely informally, with no formal semantics at all. The meaning/effect of the various constructions is explained. Example: C++.
- Part formally, part informally. Perhaps the semantics of the core constructions of the PL is described informally, while higher level constructions are described using formal semantics, in terms of the core. Example: Haskell.