CS322 Languages and Compiler Design II

Spring 2012

Lecture 4

Semantics of Programming Languages

Semantics = "Meaning"

Programming language semantics describe behavior of a language (rather than its syntax).

All Languages have informal semantics:

- e.g., "This expression is evaluated by evaluating the operator expression to obtain the function closure value, and then the argument expressions left-to-right to obtain actual parameter values, and finally executing the function with its formal parameters bound to the actual parameter values." (**fab** manual)

- Usually in English; imprecise; assumes implicit knowledge.

Idea of formal semantics:

- Describe behavior in terms of a formalism.
- To be useful, formalism should be simpler and/or better-understood than original language.
- Possible formalisms include logic, mathematical theory, abstract machines

Why bother with formal semantics?

Want a precise description of language behavior that can be used by programmer and implementor.

Formal semantics gives a machine-independent reference for correctness of implementations.

Can be used to prove properties of languages.

- E.g., Security property: a well-typed program cannot "dump core" at runtime.

- May improve language design by encouraging "cleaner" semantics (much as BNF aided language syntax design).

Varieties of Semantics

Traditionally, three rough categories:

Operational Semantics

- Describe behavior in terms of an operational model, such as an abstract machine with a specified instruction set.

Axiomatic Semantics

- Describe behavior using a logical system containing specified axioms and rules of inference.

Denotational Semantics

- Describe behavior by giving each language phrase a meaning ("denotation") in some mathematical model.

None of these approaches is entirely satisfactory (esp. compared to BNF approach to syntax). No one "best" approach – different forms may be useful for different purposes.

Syntax and Semantics

All these kinds of semantics are structured around language syntax.

Useful formalisms try to be compositional: the meaning of the whole is based on the meaning of the parts:

- semantics specifies meaning of primitive elements of the language (AST leaves)
- and of combining elements in the language (AST internal nodes)

Semantics can be described or computed by defining an attribute grammar over the language.

Operational Semantics

Define behavior of language constructs by describing how they affect the state of an abstract machine.

Abstract machine generally defined by a finite state and a set of legal state transitions (instructions).

- Like a real machine, only simpler.

Semantics is specified by giving a translation from the source language to the instruction set of the abstract machine (a compiler!)

Machine can be high-level (complicated states and instructions) or low-level (simple states and instructions).

- The lower the machine's level, the more is explained by the semantics, but the more complicated they get.
- Note similarity to choice of intermediate code level.

Operational Semantics: Simple Example

Source language AST Grammar

- Designed for easy readability

```
prog := stm
stm := stm1 ';' stm2
stm := VAR ':=' exp
stm := PRINT exp
exp := NUM
exp := VAR
exp := exp1 '+' exp2
exp := exp1 '*' exp2
```

- Ambiguity of the grammar doesn't matter, since it's for ASTs.

Very simplistic: no control flow, procedures, datatypes, etc.

Simple Abstract Machine

State =

- Stack of Values
- Global Environment mapping VARs to VALUEs
- Current Instruction Pointer (IP)

Control = List of Instructions:

`ADD`, `MULT`
: pop top two values from stack, add/multiply them, and push result

`PUSH value`
: push specified value onto stack

`FETCH var`
: fetch value of specified var from environment and push onto stack

`STORE var`
: pop top value from stack and store into specified var

`PRINT`
: pop top value from stack and print it

`HALT`

Initially: empty stack & environment; IP at start of list

SYNTAX-DIRECTED SEMANTICS DEFINITION

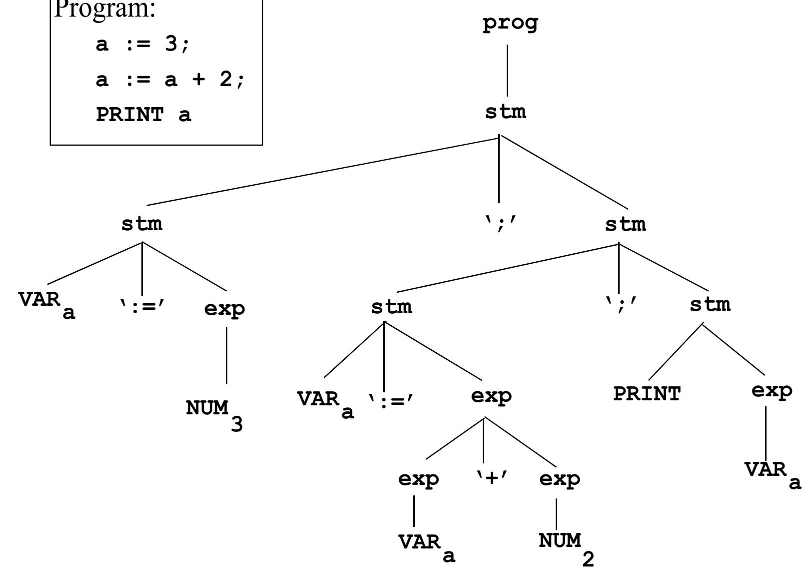
Use (synthesized) attributes to build list of instructions.

Notation: $[x_1, \ldots, x_n]$ is the list containing elements $x_1, \ldots, x_n$ and x@y is the concatenation of lists x and y.

prog := stm	prog.p := stm.p @ [HALT]
stm := stm_1 ';' stm_2	stm.p := stm_1.p @ stm_2.p
stm := VAR ':='exp	stm.p := exp.p @ [STORE VAR.var]
stm := PRINT exp	stm.p := exp.p @ [PRINT]
exp := NUM	exp.p := [PUSH NUM.num]
exp := VAR	exp.p := [FETCH VAR.var]
exp := exp_1 '+' exp_2	exp.p := exp_1.p @ exp_2.p @ [ADD]
exp := exp_1 '*' exp_2	exp.p := exp_1.p @ exp_2.p @ [MULT]

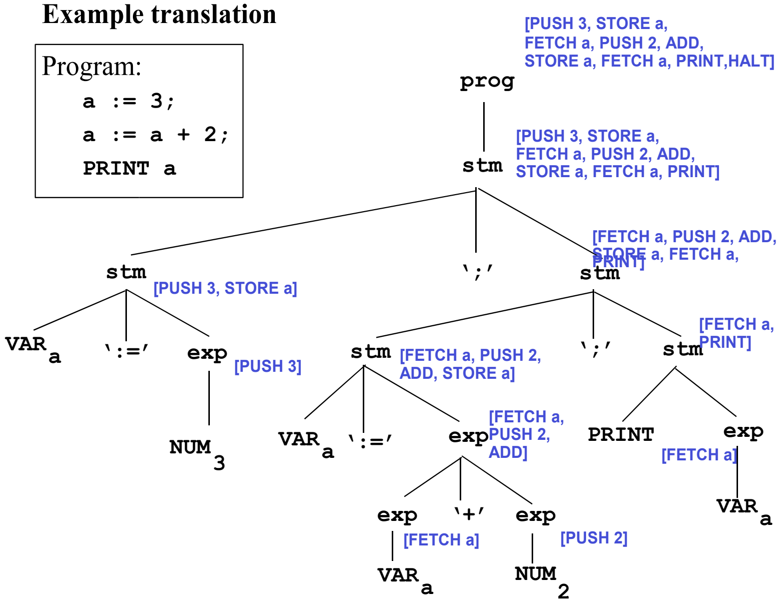
EXAMPLE PROGRAM

Program:
```
a := 3;
a := a + 2;
PRINT a
```



Example translation

Program:
```
a := 3;
a := a + 2;
PRINT a
```



SAMPLE EXECUTION

Instructions	Stack	Environment
	-	{}
PUSH 3	3	{}
STORE a	-	{a = 3}
FETCH a	3	{a = 3}
PUSH 2	3,2	{a = 3}
ADD	5	{a = 3}
STORE a	-	{a = 5}
FETCH a	5	{a = 5}
PRINT	-	{a = 5} prints 5 !!
HALT	-	{a = 5}

Example Proof using Operational Semantics

Theorem: The stack never underflows

Lemma 1: If the stack has initial size n, then the net effect of executing the instructions corresponding to an **expression** is to increase the stack size to $n+1$. Moreover, at no point during such execution is the stack size $< n$.
• Proof: By induction. `NUM` and `VAR` are base cases; + and ∗ are inductive cases.

Lemma 2: If the stack has initial size n, then the net effect of executing the instructions corresponding to a **statement** is to leave the stack at size n. Moreover, at no point during such execution is the stack size $< n$.
• Proof: By induction, with aid of Lemma 1. `PRINT` and `:=` are the base cases; `';'` is the inductive case.

Proof of theorem: Since the program starts with a stack of size 0 and executes a single statement, Lemma 2 proves that the stack never has size < 0.

Axiomatic Semantics

Describe language in terms of assertions about how statements affect predicates on program variables.

The assertion

$\{P\}\ S\ \{Q\}$

says that if P is true before the execution of S, then Q will be true after the execution of S.

Examples:

$\{y \geq 3\}$ `x := y + 1` $\{x \geq 4\}$

$\{y = 0 \wedge x = c\}$
```
   while x > 0 do
     y := y + 1;
     x := x - 1
   end
```
$\{x = 0 \wedge y = c\}$

Axioms and Rules of Inference

Axioms are simple assertions guaranteed to be true in the language, e.g.:

$\{P[y/x]\}$ `x := y` $\{P\}$

where $P[y/x]$ means P with every instance of x replaced by y.

Rules of inference are rules for deriving a true assertion from other true assertions, e.g.:

$$\frac{\{P\}S\{Q\} \quad \{Q\}T\{R\}}{\{P\}\ S;T\ \{R\}}$$

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{P \wedge \overline{B}\}}$$

Uses of Axiomatic Semantics

May be used for proving that a program implements a specification

• i.e. given axioms and rules of inference of the language, show that a given assertion about a given program is true.

Example: Prove

$\{\texttt{k} > 0\}\ \texttt{Prog}\ \{\texttt{sum} = \sum_{n=1}^{\texttt{k}} n\}$

• where `Prog` is

```
i := k; sum := k;
while i > 1 do
   i := i - 1;
   sum := sum + i
end;
```

• Can be done by repeated application of axioms and rules. Axiomatic methods become somewhat unwieldy in presence of side- effects and aliasing (multiple names for one storage location). For handling real programs, automated ”proof assistant” is essential.

Example Proof of correctness in annotation form

$\{\mathtt{k} > 0\}$
$\{\mathtt{k} = \sum_{n=\mathtt{k}}^{\mathtt{k}} n \wedge \mathtt{k} > 0\}$
`i := k;`
$\{\mathtt{k} = \sum_{n=\mathtt{i}}^{\mathtt{k}} n \wedge \mathtt{i} > 0\}$
`sum := k;`
$\{\mathtt{sum} = \sum_{n=\mathtt{i}}^{\mathtt{k}} n \wedge \mathtt{i} > 0\}$
`while i > 1 do`
    $\{\mathtt{sum} = \sum_{n=\mathtt{i}}^{\mathtt{k}} n \wedge \mathtt{i} > 0 \wedge \mathtt{i} > 1\}$
    `i := i - 1;`
    $\{\mathtt{sum} = \sum_{n=\mathtt{i}+1}^{\mathtt{k}} n \wedge \mathtt{i} > 0\}$
    `sum := sum + i`
    $\{\mathtt{sum} = \sum_{n=\mathtt{i}}^{\mathtt{k}} n \wedge \mathtt{i} > 0\}$
`end;`
$\{\mathtt{sum} = \sum_{n=\mathtt{i}}^{\mathtt{k}} n \wedge \mathtt{i} > 0 \wedge \overline{\mathtt{i} > 1}\}$
$\{\mathtt{sum} = \sum_{n=1}^{\mathtt{k}} n\}$

Denotational Semantics

Program statements and expressions denote mathematical functions between abstract semantic domains.

• In particular, the program as a whole denotes a function from some domain of inputs to some domain of answers.

Semantics are specified as a set of denotation functions mapping pieces of program syntax to suitable mathematical functions.

• Functions are attached to corresponding grammatical constructs using synthesized attribute grammars.

Proper definition of semantic domains is complicated subject – we'll ignore.

Common notation: $\lambda x.e$ is an anonymous function with argument x and body e.

`λx.x+1     λy.if y < 0 then -y else y`

Denotational Semantics of Straight-line Programs

Semantic domains:

```
V = Int (values)
Ide     (identifiers)
S = Ide → V    (stores)
Exp = S → V    (expressions)
Stm = S → S    (statements)
```

Denotation functions (from syntactic class to semantic domain):

```
I: ID → Ide
N: NUM → V
E: exp → Exp
S: stm → Stm
```

Auxiliary functions:

```
plus: V × V → V
update: (S × Ide × V) → S
```


Denotation functions

stm → ID := exp	S[stm] = λs.update(s,I[ID],E[exp]s)
stm → $\mathtt{stm}_1$;$\mathtt{stm}_2$	S[stm] = λs.S[$\mathtt{stm}_2$](S[$\mathtt{stm}_1$]s)
exp → NUM	E[exp] = λs.N[NUM]
exp → ID	E[exp] = λs.s(I[ID])
exp → $\mathtt{exp}_1$ + $\mathtt{exp}_2$	E[exp] = λs.plus(E[$\mathtt{exp}_1$]s,E[$\mathtt{exp}_2$]s)
exp → ($\mathtt{exp}_1$)	E[exp] = E[$\mathtt{exp}_1$]
	N[NUM] = NUM.num
	I[ID] = ID.ident

Facts about stores and updates

Definition of update:

$$\texttt{update} = \lambda(s, id, v).\ \lambda id_1.\texttt{if}\ id = id_1\ \texttt{then}\ v\ \texttt{else}\ s\ id_1$$

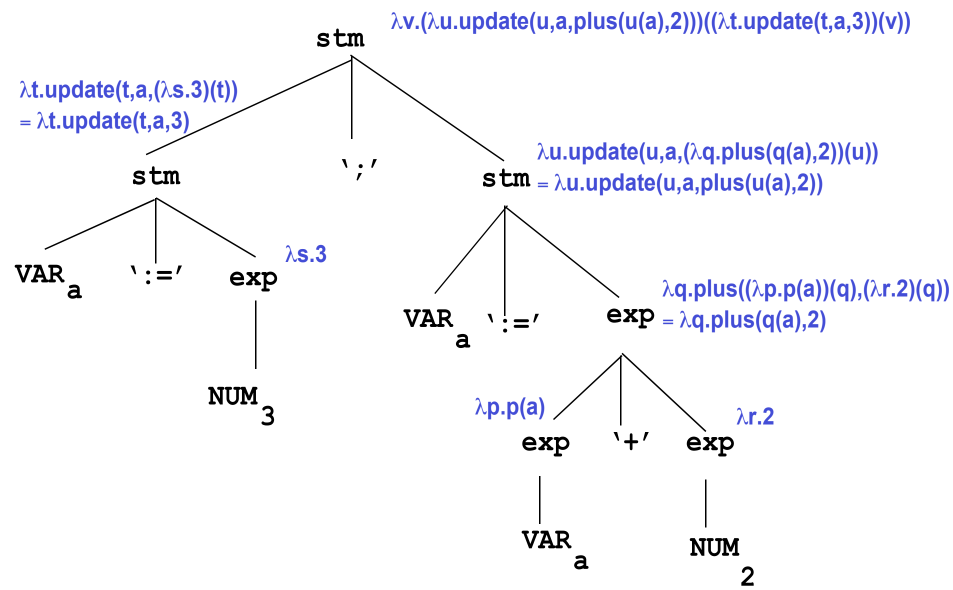
Fact A: For any s_0, x, i:

$$(\texttt{update}(s_0, x, i))\ x$$
$$= (\lambda id_1.\texttt{if}\ x = id_1\ \texttt{then}\ i\ \texttt{else}\ s_0\ id_1)\ x$$
$$= (\texttt{if}\ x = x\ \texttt{then}\ i\ \texttt{else}\ s_0\ x)$$
$$= i$$

Fact B: For any s_0, x, i, j :

$$\texttt{update}(\texttt{update}(s_0, x, i), x, j) = \texttt{update}(s_0, x, j)$$

Calculating the Denotation of a Program



Simplifying Denotation

$$\lambda\texttt{v}.(\lambda\texttt{u}.\texttt{update(u,a,plus(u(a),2))})((\lambda\texttt{t}.\texttt{update(t,a,3)})(\texttt{v}))$$
$$= \lambda\texttt{v}.(\lambda\texttt{u}.\texttt{update(u,a,plus(u(a),2))})(\texttt{update(v,a,3)})$$
$$= \lambda\texttt{v}.\texttt{update(update(v,a,3),a,plus((update(v,a,3))(a),2))}$$
$$= \lambda\texttt{v}.\texttt{update(update(v,a,3),a,plus(3,2))} \quad \text{(using Fact A)}$$
$$= \lambda\texttt{v}.\texttt{update(update(v,a,3),a,5)}$$
$$= \lambda\texttt{v}.\texttt{update(v,a,5)} \quad \text{(using Fact B)}$$