CS302 Languages and Compiler Design II

Lecture 3

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., "If it is followed by a (possibly empty) parameter list, the designator implies an activation of the procedure and stands for the value resulting from its execution" (Oberon Report Par. 8.1)
- 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).

Kinds of Semantics

Three traditional 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: [x1,...,xn] is the list containing elements x1,...,xn and x@y is the concatenation of lists x and y.

```
prog := stm
                prog.p := stm.p @ [HALT]
stm := stm1 ';' stm2
                stm.p := stm1.p @ stm2.p
stm := VAR ':='exp
                stm.p := exp.p @ [STORE VAR.var]
stm := PRINT exp
                stm.p := exp.p @ [PRINT]
```

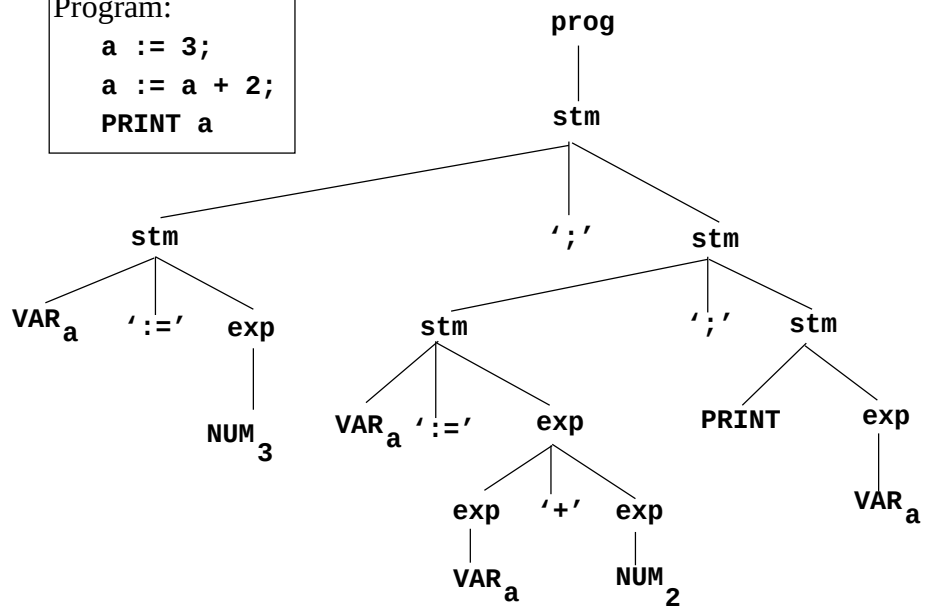

Syntax-directed definition (continued)

```
exp := NUM
                          exp.p := [PUSH NUM.num]
exp := VAR
                        exp.p := [FETCH VAR.var]
exp := exp1 '+' exp2
              exp.p := exp1.p @ exp2.p @ [ADD]
exp := exp1 '*' exp2
             exp.p := exp1.p @ exp2.p @ [MULT]
```

Example Program

Program:

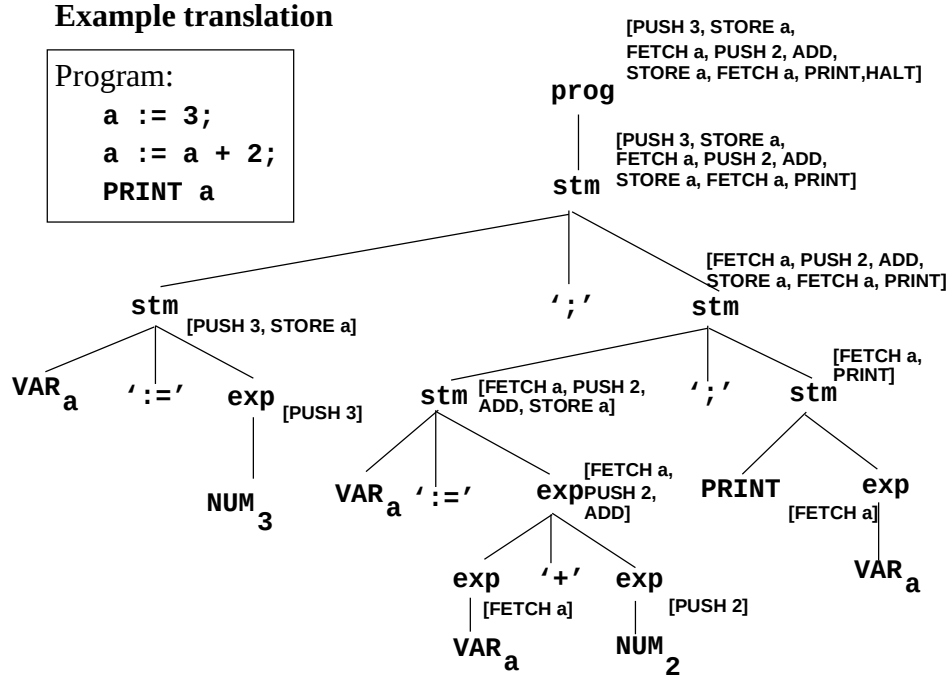
```
a := 3;
a := a + 2;
PRINT a
```



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 of 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 ≥ 3} x := y + 1  {x ≥ 4}
{y = 0 ^ x = c} while x > 0 do
                          y := y + 1;
                          x := x - 1
                 end {x = 0 ^ 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.:

```
{P} S {Q}   {Q} T {R}
_____________________
    {P} S;T {R}

     {P ^ B} S {P}
_____________________________
{P} while B do S {P ^ ~B}
```

Uses of Axiomatic Semantics

Often used for proving programs "correct".

- I.e., given, axioms and rules of inference of the language, show that a given assertion about a given program is true.
- Example: Prove

```
{k > 0} Prog {sum = 1 + 2 + ... + k}
```

- 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 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

```
{k > 0}
{k = k + … + k ^ k > 0}
i := k;
{k = i + … + k ^ i > 0}
sum := k;
{sum = i + … + k ^ i > 0}
while i > 1 do
   {sum = i + … + k ^ i > 0 ^ i > 1}
    i := i - 1;
   {sum = (i+1) + … + k ^ i > 0}
   sum := sum + i
   {sum = i + … + k ^ i > 0}
end;
{sum = i + … + k ^ i > 0 ^ ~(i > 1)}
{sum = 1 + … + k}
```

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: λ**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 x V -> V
update: (S x Ide x V) -> S
```

Denotation function definition

```
stm -> ID := exp
        S[stm] = λs.update(s,I[ID],E[exp]s)
stm -> stm1; stm2
              S[stm] = λs.S[stm2](S[stm1]s)
exp -> NUM
                           E[exp] = λs.N[NUM]
exp -> ID
                         E[exp] = λs.s(I[ID])
exp -> exp1 + exp2
        E[exp] = λs.plus(E[exp1]s,E[exp2]s)
exp -> (exp1)
                              E[exp] = E[exp1]
N[NUM] = NUM.num
I[ID] = ID.ident
```


Facts about stores and updates

Definition of update:

```
update = λ(s,id,v).
         λid1.if id = id1 then v else s id1
```

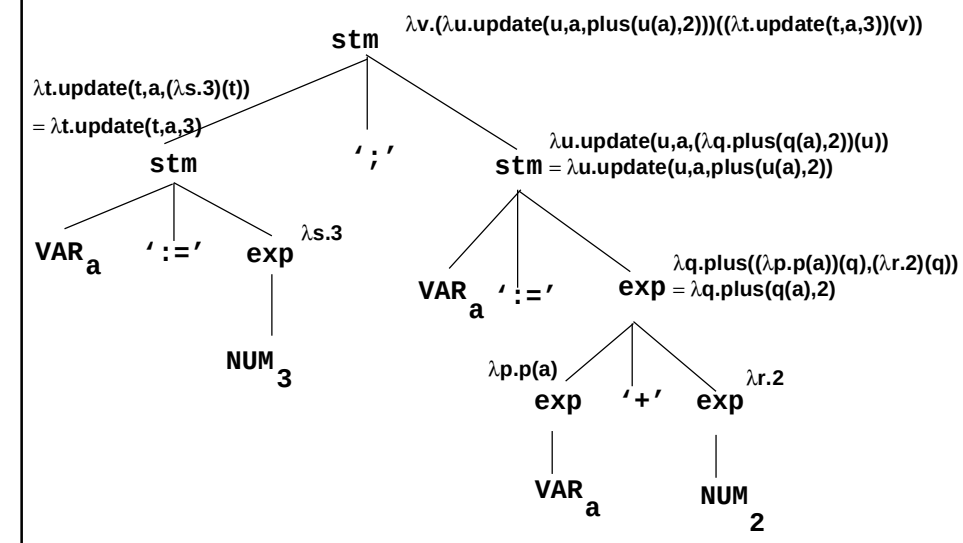
Fact A: For any `s0,x,i:`

```
  (update(s0,x,i)) x
   = (λid1.if x = id1 then i else s0 id1) x
   = (if x = x then i else s0 x)
   = i
```

Fact B: For any `s0,x,i,j :`

```
   update(update(s0,x,i),x,j) =
       update(s0,x,j)
```

Calculating the denotation of a program.



Simplifying denotation:

λv.(λu.update(u,a,plus(u(a),2)))((λt.update(t,a,3))(v))
= λv.(λu.update(u,a,plus(u(a),2)))(update(v,a,3))
= λv.update(update(v,a,3),a,plus((update(v,a,3))(a),2))
= λv.update(update(v,a,3),a,plus(3,2)) (using Fact A)
= λv.update(update(v,a,3),a,5)
= λv.update(v,a,5) (using Fact B)