# Formal Semantics I: Operational Semantics

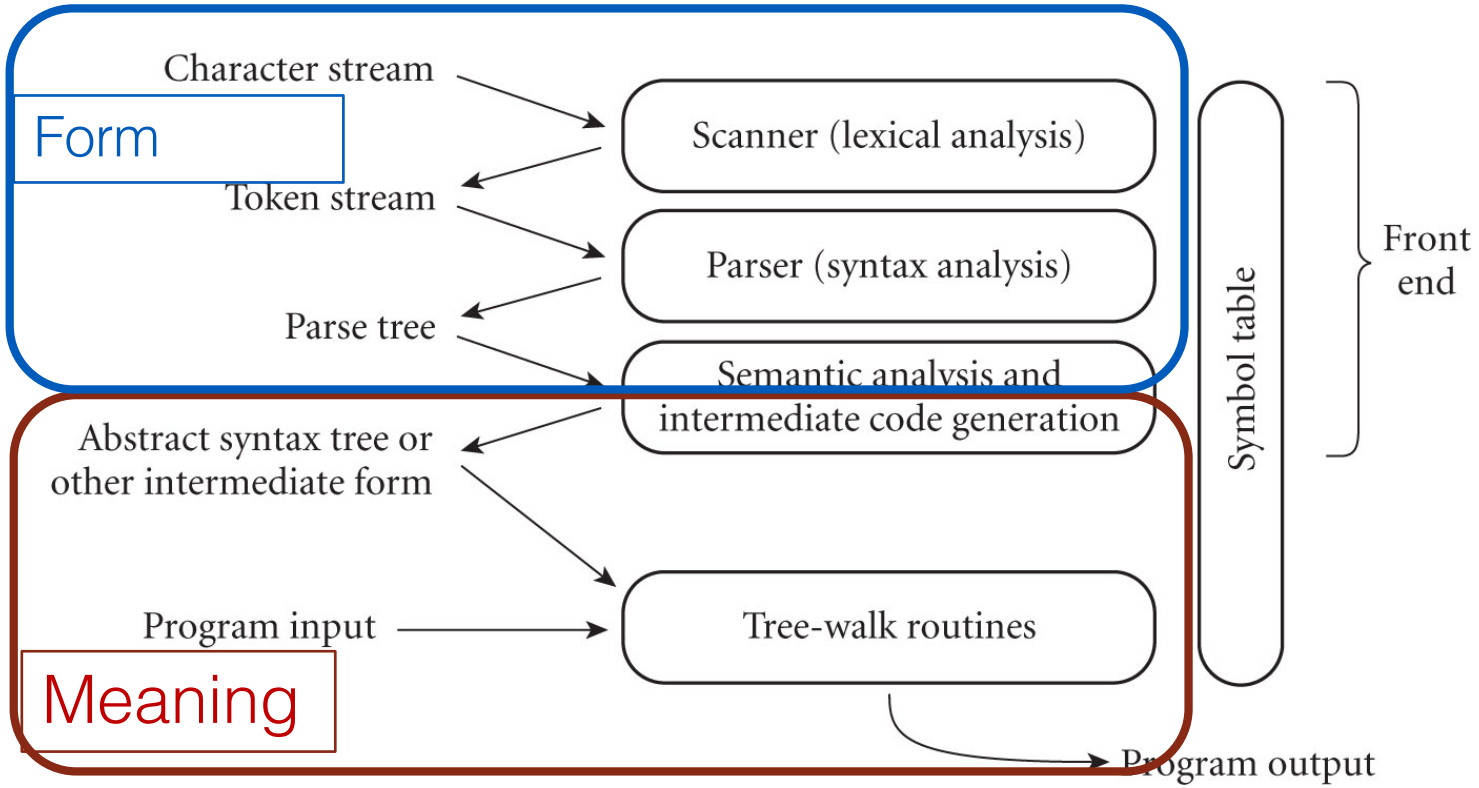
## CAS CS 320: Principles of Programming Languages

Thursday, March 28, 2024

# Administrivia

- Homework 7 due Friday, Mar 29 (tomorrow), by 11:59 pm.
- Homework 8 posted Friday, Mar 29 (tomorrow), and due Friday, April 5.
- Withdraw deadline (grade W) is Friday, Mar 29.

# Form and Meaning



# Syntax vs Semantics

- Syntax is about "form" and semantics about "meaning".

```
<expr> ::= <expr> <addop> <term>
           | <term>
<term> ::= <term> <mulop> <term>
           | <factor>
<factor> ::= <const> | ( <expr> )
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>   ::= + | -
<mulop>   ::= * | /
```

- How do we give meaning to sentences from this grammar?

We have two kinds of semantics: static and dynamic

# Semantics: Static vs Dynamic

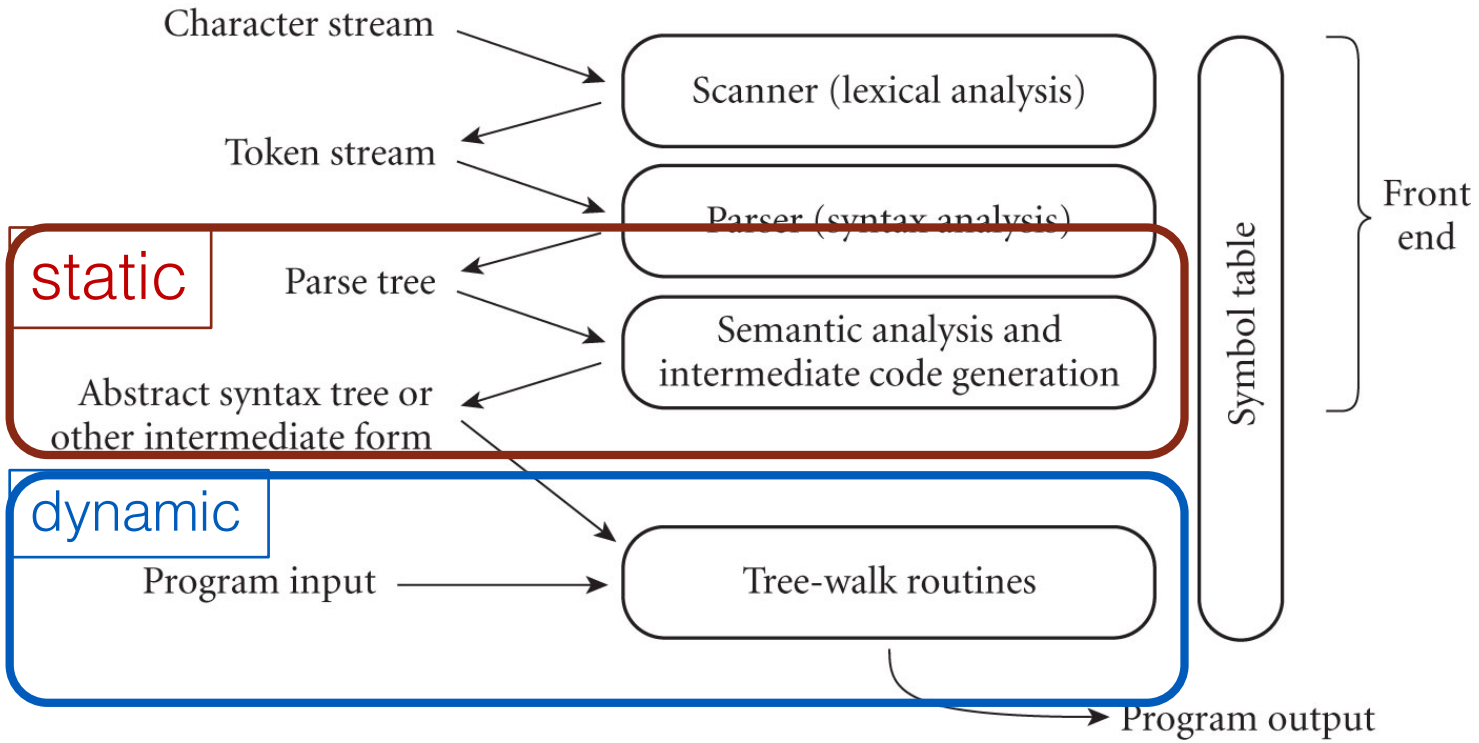
**Static semantics:**

- Set of rules attaching some high level meaning to the syntactic structure.
  Examples: typing information, well-formedness of commands, etc.
- It is usually enforced statically (before execution).

**Dynamic semantics:**

- Set of rules describing how the syntactic objects need to be executed.
  Examples: expression evaluation, commands execution, etc.
- It described the way the program must be executed at runtime.

# Parsing and semantic analysis



# Dynamic Semantics

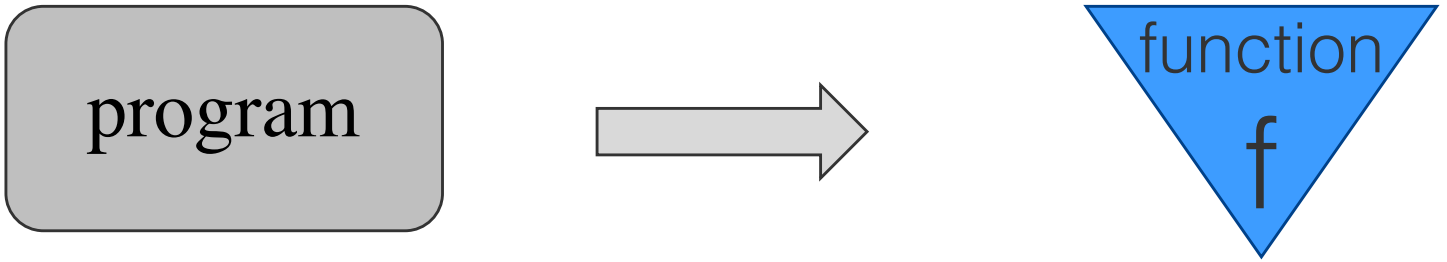
## Why do we need to specify the semantics?

- Programmers need to know what statements and command mean
- Compiler writers must know exactly what language constructs do
- Correctness proofs with respect to the specifications,
- Designers need to detect ambiguities and inconsistencies
- …

How would you specify the semantics of a program?

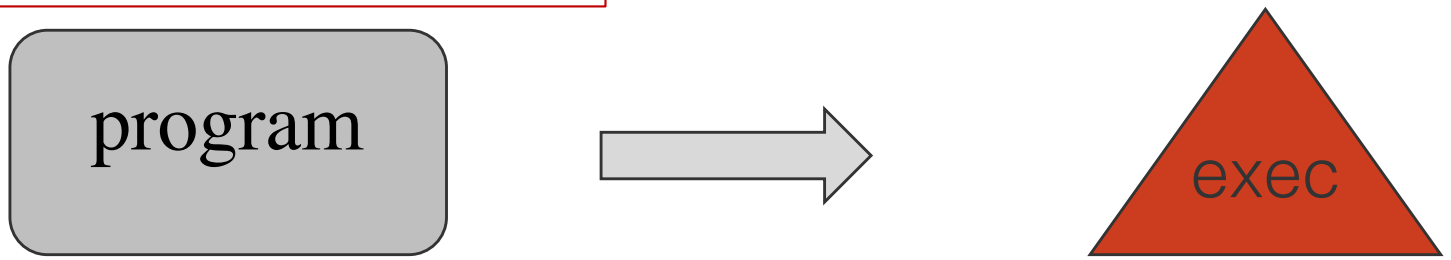
# Formal Semantics

## Denotational Semantics




A program is described by a mathematical function specifying the input-output relation that the program implements.

## Operational Semantics




A program is described by the sequence of transformations that the program implements on the input to produce the output.

(We ignore denotational semantics, though very interesting, and focus on operational semantics)

# Operational Semantics

- Gives the meaning of a program by describing how the statements are executed.
- This can be provided through formal rules or through a description of a machine to execute the programs.
- The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement.

Some examples

- https://docs.oracle.com/javase/specs/jvms/se7/html/
- http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- http://sml-family.org/sml97-defn.pdf
- Our interpreter.

# Operational Semantics

- It is usually provided at a level of abstraction that is independent from the machine.
- The detailed characteristics of the particular computer would make actions difficult to describe/understand.
- Different formalism has been developed to describe the operational semantics in a machine-independent way.

We will look into formal rules and derivations.

# Examples of Operational Semantics for Programming Languages – partly or entirely based on *abstract machines*.

An *abstract machine* (not a real physical machine) is a theoretical model that allows for a detailed and precise analysis of how programs (in a given programming language) are executed.

- The Java Virtual Machine Specification (a few hundred pages)
- Programming languages — C (several hundred pages)
- The Definition of Standard ML (136 pages)
- Our interpreter … (using *reduction rules*, implicitly an *abstract machine*)

# Language for the Interpreter (simplified)

The language for the interpreter can be described by the following grammar:

```
<const> ::= int | name
<prog>  ::= <com>;<prog> | quit
<com>   ::= push <const> | pop | add | sub | mul | div
```

- A program is a sequence of commands followed by `quit`.
- A command is one the keywords above - in the case of push this is followed by a constant.
- A (simplified) constant is either an int or a string.
- We will denote arbitrary programs with `p,p',…`

# Operational semantics for the interpreter

$$(p/S) \longrightarrow (p'/S')$$

Here $(p/S)$ is a configuration where p is a program and S is a stack. We call these pairs configurations because we think in terms of an "abstract machine".

We can think about the stack as a list of values (denoted with $v$):

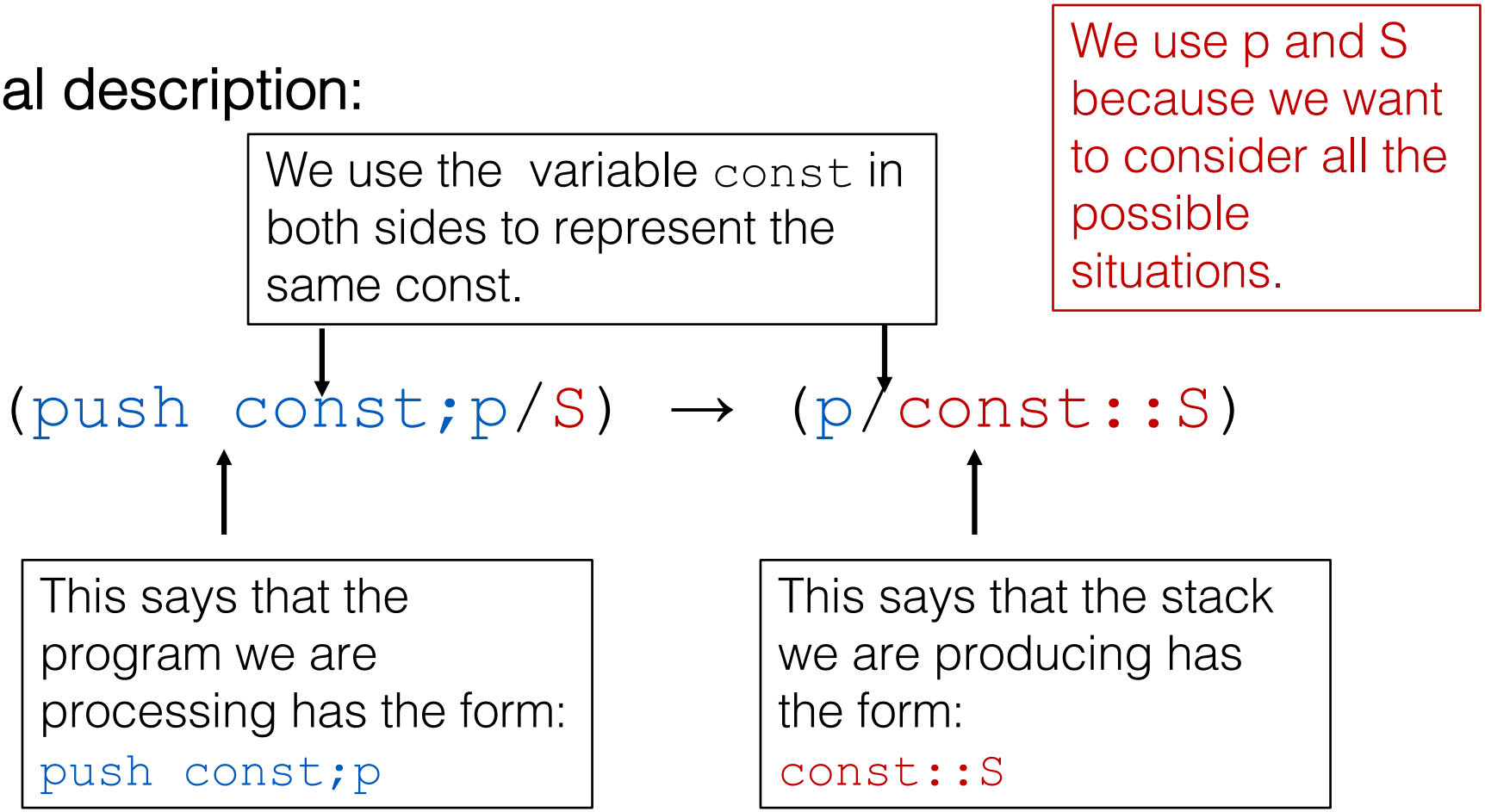
$$v_n :: \ldots :: v_2 :: v_1 :: []$$

We say that from the configuration (p/S) we can step (or reduce) to the configuration (p'/S') in one step.

# An example: `push const`

**Informal description:** Pushing `const` to the stack.

**Formal description:**



# Another example: `pop`

**Informal description:** Remove the top value from the stack.

**Formal description:**

We use p and S because we want to consider all the possible situations.

$$(\texttt{pop};\texttt{p}/\texttt{v}::\texttt{S}) \longrightarrow (\texttt{p}/\texttt{S})$$

This says that the program we are processing has the form: `pop;p`

This says that the stack we are producing has the form: `S`

Does this rule capture all the possible situations?

# Another example: `pop` from an empty stack

**Informal description:** If the stack is empty we stop and output Error.

**Formal description:**

$$(\texttt{pop;p}/[\,]) \longrightarrow \texttt{Error}$$

# Another example: `add`

Informal description: Add consumes the top two values in the stack, and pushes their sum to the stack. If there are fewer then 2 values on the stack, terminate and report error. If two top values in the stack are not integers, terminate and report error.

Formal description:

What can we do for this case?

$$(\texttt{add;p}/v_2::v_1::S) \longrightarrow (\texttt{p}/v_2+v_1::S)$$

$$(\texttt{add;p}/v_1::[]) \longrightarrow \texttt{Error}$$

$$(\texttt{add;p}/[]) \longrightarrow \texttt{Error}$$

This + represents the addition of the two values

# Revisiting the stack

**First try:** We can think about the stack as a list of values (denoted with `v`):

$$v_n :: \ldots :: v_2 :: v_1 :: []$$

What is the problem with this?

We don't distinguish values of different types.

**Second try:** We can think about the stack as a list of typed values (denoted with `type(v)`):

$$\texttt{int}(v_n) :: \ldots :: \texttt{name}\ (v_2) :: \texttt{int}(v_1) :: []$$

# Revisiting `add`

Informal description: Add consumes the top two values in the stack, and pushes their sum to the stack. If there are fewer then 2 values on the stack, terminate and report error. If two top values in the stack are not integers, terminate and report error.

Formal description:

$(\texttt{add;p/int}(v_2)\texttt{::int}(v_1)\texttt{::S}) \rightarrow (\texttt{p/int}(v_2+v_1)\texttt{::S})$

$(\texttt{add;p/}v_1\texttt{::[]}) \rightarrow \texttt{Error}$

$(\texttt{add;p/[]}) \rightarrow \texttt{Error}$

$(\texttt{add;p/name}(v)\texttt{::S}) \rightarrow \texttt{Error}$

$(\texttt{add;p/int}(v_1)\texttt{::name}(v_2)\texttt{::S}) \rightarrow \texttt{Error}$

Are we done?

( THIS PAGE INTENTIONALLY LEFT BLANK )