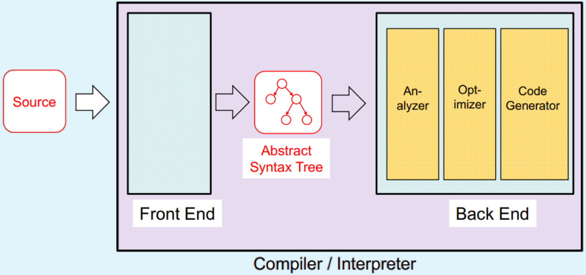
CENG 2010 - Programming Language Concepts

Week 5: (Operational) Semantics

Burak Ekici

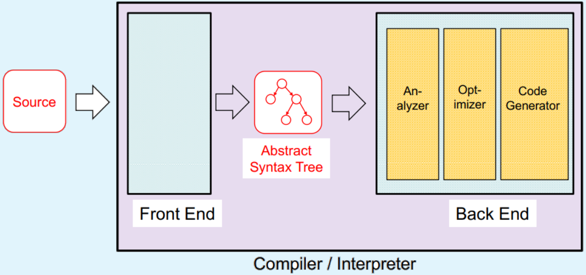
April 3 - April 10, 2023

Architecture of Compilers and Interpreters



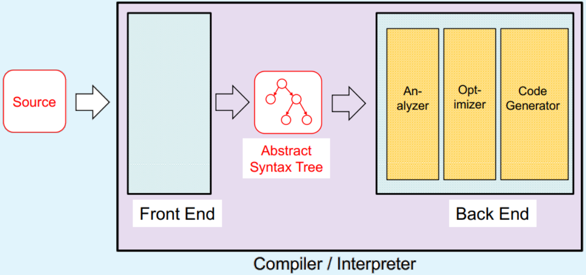
lexer	:	source code	$\mapsto$	tokens (keywords, variables, numbers, etc.)	regular expressions

Architecture of Compilers and Interpreters



lexer	:	source code	$\mapsto$	tokens (keywords, variables, numbers, etc.)	regular expressions
parser	:	tokens	$\mapsto$	abstract syntax trees(AST)/parse trees	context free grammars

Architecture of Compilers and Interpreters



lexer	:	source code	$\mapsto$	tokens (keywords, variables, numbers, etc.)	regular expressions
parser	:	tokens	$\mapsto$	abstract syntax trees(AST)/parse trees	context free grammars
code generator	:	AST	$\mapsto$	intermediate (OCaml) code	

Table of Contents

Syntax vs Semantics

- syntax vs semantics

syntax	grammatical structure
semantics	underlying meaning

Syntax vs Semantics

- syntax vs semantics

syntax	grammatical structure
semantics	underlying meaning

- similar semantics can be achieved by different syntax in different languages

	Physical Equality	Structural Equality
C	$\&a == \&b$	*a == *b
OCaml	$a == b$	$a = b$

Syntax vs Semantics

- syntax vs semantics

syntax	grammatical structure
semantics	underlying meaning

- similar semantics can be achieved by different syntax in different languages

	Physical Equality	Structural Equality
C	$\&$a $==$ $\&$b	*a $==$ *b
OCaml	a $==$ b	a $=$ b

- semantics in prose text vs formal semantics in mathematics

Formal Semantics of a Programming Language

- mathematical description of the meaning of programs written in that language

Formal Semantics of a Programming Language

- mathematical description of the meaning of programs written in that language
- main approaches to formal semantics

denotational	algebraic objects
operational	abstract machines
axiomatic	logical transformations

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
  - convert programs into functions/functors mapping inputs to outputs

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
  - convert programs into functions/functors mapping inputs to outputs
- operational semantics: define “how programs execute”

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
  - convert programs into functions/functors mapping inputs to outputs
- operational semantics: define "how programs execute"
  - often on an abstract machine (mathematical model of computer)

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
    - convert programs into functions/functors mapping inputs to outputs
- operational semantics: define “how programs execute”
    - often on an abstract machine (mathematical model of computer)
- axiomatic semantics: describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
  - convert programs into functions/functors mapping inputs to outputs
- operational semantics: define “how programs execute”
  - often on an abstract machine (mathematical model of computer)
- axiomatic semantics: describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
  - pre-conditions: assumed properties of initial states

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
  - convert programs into functions/functors mapping inputs to outputs
- operational semantics: define “how programs execute”
  - often on an abstract machine (mathematical model of computer)
- axiomatic semantics: describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
  - pre-conditions: assumed properties of initial states
  - post-condition: guaranteed properties of final states

Styles of Formal Semantics

- denotational semantics: represent programs with mathematical objects
  - convert programs into functions/functors mapping inputs to outputs
- operational semantics: define "how programs execute"
  - often on an abstract machine (mathematical model of computer)
- axiomatic semantics: describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
  - pre-conditions: assumed properties of initial states
  - post-condition: guaranteed properties of final states
  - logical rules describe how to systematically build up these transformers from programs

Operational Semantics

- describe the evaluation of programs on an abstract machine

Operational Semantics

- describe the evaluation of programs on an abstract machine
- approach: benefiting rules in the form of judgments $\dfrac{H_1 \ldots H_n}{C}$

Operational Semantics

- describe the evaluation of programs on an abstract machine
- approach: benefiting rules in the form of judgments $\dfrac{H_1 \dots H_n}{C}$
- if the conditions $H_1 \dots H_n$ ("hypotheses") hold, then the condition C ("conclusion") holds

Operational Semantics

- describe the evaluation of programs on an abstract machine
- approach: benefiting rules in the form of judgments $\dfrac{H_1 \dots H_n}{C}$
- if the conditions $H_1 \dots H_n$ ("hypotheses") hold, then the condition C ("conclusion") holds
- if $n = 0$ (no hypotheses) then the conclusion automatically holds: an axiom

Operational Semantics

- describe the evaluation of programs on an abstract machine
- approach: benefiting rules in the form of judgments $\dfrac{H_1 \ldots H_n}{C}$
- if the conditions $H_1 \ldots H_n$ ("hypotheses") hold, then the condition C ("conclusion") holds
- if $n = 0$ (no hypotheses) then the conclusion automatically holds: an axiom
- inference rules let one to speak about the evaluation steps of a given expression

Operational Semantics

- describe the evaluation of programs on an abstract machine
- approach: benefiting rules in the form of judgments $\dfrac{H_1 \ldots H_n}{C}$
- if the conditions $H_1 \ldots H_n$ ("hypotheses") hold, then the condition C ("conclusion") holds
- if $n = 0$ (no hypotheses) then the conclusion automatically holds: an axiom
- inference rules let one to speak about the evaluation steps of a given expression
- judgments are of

  $(e, \Gamma) \rightsquigarrow e$ expressions

  $(c, \Gamma) \rightsquigarrow \Gamma$ commands

  form that could be represented as OCaml functions

Operational Semantics

- describe the evaluation of programs on an abstract machine
- approach: benefiting rules in the form of judgments $\dfrac{H_1 \ldots H_n}{C}$
- if the conditions $H_1 \ldots H_n$ ("hypotheses") hold, then the condition C ("conclusion") holds
- if $n = 0$ (no hypotheses) then the conclusion automatically holds: an axiom
- inference rules let one to speak about the evaluation steps of a given expression
- judgments are of

  $$\begin{array}{llll} (e, \Gamma) & \rightsquigarrow & e & \text{expressions} \\ (c, \Gamma) & \rightsquigarrow & \Gamma & \text{commands} \end{array}$$

  form that could be represented as OCaml functions
- this way of presenting the semantics is handled by a definitional interpreter

Table of Contents

The IMP Language (Abstract Syntax)

- a prototypical imperative language with structured control flow

The IMP Language (Abstract Syntax)

- a prototypical imperative language with structured control flow
- composed of expressions (arithmetic, Boolean) and commands

$$
\begin{aligned}
a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid a_1 / a_2 \\
b &::= \texttt{true} \mid \texttt{false} \mid b_1 \mathbin{\&\&} b_2 \mid b_1 \mathbin{||} b_2 \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 < a_2 \mid \sim b_1 \\
c &::= \texttt{skip} \mid \texttt{int}\ x \mid x := a \mid c_1 ; c_2 \mid \texttt{if}(b)\texttt{then}\{c_1\}\texttt{else}\{c_2\} \mid \texttt{while}(b)\{c\}
\end{aligned}
$$

Example (An IMP Program – Factorial)

```
// input: an arbitrary integer in a

      int a;
      int res;
      res := 1;
      a := 6;
      while(a > 1)
      {
         res := a * res;
         a   := a - 1
      }

// output: a! in res
```

Example (An IMP Program – Division)

```
// input: dividend in a, divisor in b

      int a;
      int b;
      int q;
      int r;
      a := 100;
      b := 3;
      r := a;
      q := 0;
      while(b < r || b = r)
      {
         r := r − b;
         q := q + 1
      }

// output: quotient in q, remainder in r
```

The IMP Language (Arithmetic Expressions)

a language containing

The IMP Language (Arithmetic Expressions)

a language containing

- integer constants/values (v) $\quad \ldots, -5, 0, 1, 5, \ldots$

The IMP Language (Arithmetic Expressions)

a language containing

- integer constants/values (v) $\ldots, -5, 0, 1, 5, \ldots$
- variables $x, y, \ldots$

The IMP Language (Arithmetic Expressions)

a language containing

- integer constants/values (v) $\ldots, -5, 0, 1, 5, \ldots$
- variables $x, y, \ldots$
- operations $e_1 + e_2$, $e_1 - e_2$, $e_1 \times e_2$ and e_1 / e_2

  where e_1, e_2 arithmetic expressions

The IMP Language (Arithmetic Expressions)

a language containing

- integer constants/values (v) $\ldots, -5, 0, 1, 5, \ldots$
- variables $x, y, \ldots$
- operations $e_1 + e_2$, $e_1 - e_2$, $e_1 \times e_2$ and e_1 / e_2

  where e_1, e_2 arithmetic expressions

Implementation (OCaml)

```
type aexpr =
  | Aconst: int           → aexpr
  | Var   : string        → aexpr
  | Plus  : (aexpr∗aexpr) → aexpr
  | Minus : (aexpr∗aexpr) → aexpr
  | Mult  : (aexpr∗aexpr) → aexpr
  | Div   : (aexpr∗aexpr) → aexpr
```

The IMP Language (Boolean Expressions)

a language comprising

The IMP Language (Boolean Expressions)

a language comprising

- boolean constants/values (v) `true` and `false`

The IMP Language (Boolean Expressions)

a language comprising

- boolean constants/values (v) `true` and `false`
- boolean operators $b_1 \&\& b_2,\ b_1 || b_2,\ a_1 = a_2,\ a_1 > a_2,\ a_1 < a_2\ \sim b_1$

  where a_1, a_2 arithmetic expressions;
  b_1, b_2 boolean expressions

The IMP Language (Boolean Expressions)

a language comprising

- boolean constants/values (v) `true` and `false`
- boolean operators $b_1 \&\& b_2,\ b_1 || b_2,\ a_1 = a_2,\ a_1 > a_2,\ a_1 < a_2\ \sim b_1$

where a_1, a_2 arithmetic expressions; b_1, b_2 boolean expressions

Implementation (OCaml)

```
type bexpr =
  | Bconst: bool          → bexpr
  | And   : (bexpr∗bexpr) → bexpr
  | Or    : (bexpr∗bexpr) → bexpr
  | Eq    : (aexpr∗aexpr) → bexpr
  | Gt    : (aexpr∗aexpr) → bexpr
  | Lt    : (aexpr∗aexpr) → bexpr
  | Neg   : bexpr         → bexpr
```

The IMP Language (Commands)

a language made of

`skip`	do nothing	
`int` x	variable declaration and initialization (set to 0)	integer type only
$x := a$	variable assignment	arithmetic expressions only
$c_1 ; c_2$	sequencing	
$\texttt{if}(b)\texttt{then}\{c_1\}\texttt{else}\{c_2\}$	branching	
$\texttt{while}(b)\{c\}$	looping	

The IMP Language (Commands)

a language made of

`skip`	do nothing	
`int` x	variable declaration and initialization (set to 0)	integer type only
$x := a$	variable assignment	arithmetic expressions only
$c_1; c_2$	sequencing	
`if`(b)`then{`c_1`}else{`c_2`}`	branching	
`while`(b)`{`c`}`	looping	

Implementation (OCaml)

```
type cmd =
  | Skip     : cmd
  | Declare  : string             → cmd
  | Assign   : (string∗aexpr)     → cmd
  | Sequence : (cmd∗cmd)          → cmd
  | Ite      : (bexpr∗cmd∗cmd)    → cmd
  | While    : (bexpr∗cmd)        → cmd
```

Program State

- IMP allows for imperative updates

Program State

- IMP allows for imperative updates
- maintains a program state (informally) = external memory

Program State

- IMP allows for imperative updates
- maintains a program state (informally) = external memory
- state (formally) = a list of variable-value pairs:

$$
\begin{array}{lll}
\Gamma := & & \\
\quad | & [\,] & \text{“empty” state} \\
\quad | & \Gamma, (x, n) & \text{“non-empty” state}
\end{array}
$$

Program State

- IMP allows for imperative updates
- maintains a program state (informally) = external memory
- state (formally) = a list of variable-value pairs:

$$
\begin{array}{lll}
\Gamma ::= & & \\
\mid & [\,] & \text{“empty” state} \\
\mid & \Gamma, (x, n) & \text{“non-empty” state}
\end{array}
$$

- captures variable declaration and initializations

Program State

- IMP allows for imperative updates
- maintains a program state (informally) = external memory
- state (formally) = a list of variable-value pairs:

$$
\begin{array}{lll}
\Gamma := & & \\
\quad | & [\,] & \text{“empty” state} \\
\quad | & \Gamma, (x, n) & \text{“non-empty” state}
\end{array}
$$

- captures variable declaration and initializations
- interface functions:

$$
\begin{array}{lll}
\text{lookup} & : & \Gamma \rightarrow \text{variable} \rightarrow \mathbb{Z} \\
\text{update} & : & \Gamma \rightarrow \text{variable} \rightarrow \mathbb{Z} \rightarrow \Gamma \\
\text{extend} & : & \Gamma \rightarrow \text{variable} \rightarrow \mathbb{Z} \rightarrow \Gamma
\end{array}
$$

Implementation (OCaml)

```
type state = (string*int) list

let extend(m: state) (s: string) (v: int): state = (s, v) :: m

let rec update(m: state) (s: string) (v: int): state =
  match m with
  | []           → extend m s v
  | (x, y) :: r → if x = s then (x, v) :: r else (x, y) :: (update r s v)

let rec lookup(m: state) (s: string): int =
  match m with
  | []            → failwith "unknown␣variable"
  | (x, y) :: xs → if x = s then y else (lookup xs s)
```

Table of Contents

The IMP Language (Small-step Operational Semantics – Arithmetic and Boolean Expressions)

$\rightsquigarrow_a \quad : \Gamma \rightarrow \text{arithmetic expression} \rightarrow \text{arithmetic expression}$

The IMP Language (Small-step Operational Semantics – Arithmetic and Boolean Expressions)

$$\begin{aligned} \rightsquigarrow_a &: \Gamma \to \text{arithmetic expression} \to \text{arithmetic expression} \\ \rightsquigarrow_b &: \Gamma \to \text{boolean expression} \to \text{boolean expression} \end{aligned}$$

The IMP Language (Small-step Operational Semantics – Arithmetic and Boolean Expressions)

$\rightsquigarrow_a \;:\; \Gamma \rightarrow \text{arithmetic expression} \rightarrow \text{arithmetic expression}$

$\rightsquigarrow_b \;:\; \Gamma \rightarrow \text{boolean expression} \rightarrow \text{boolean expression}$

$$\frac{}{\Gamma, v \rightsquigarrow_a v}\ \text{(const)} \qquad \frac{\text{lookup } \Gamma\ x = v}{\Gamma, x \rightsquigarrow_a v}\ \text{(var)} \qquad \frac{\Gamma, a_1 \rightsquigarrow_a v_1 \qquad \Gamma, a_2 \rightsquigarrow_a v_2}{\Gamma, (a_1 \text{ op } a_2) \rightsquigarrow_a (v_1 \text{ op}_{\mathbb{Z}}\ v_2)}\ \text{(aops)}$$

$$\frac{}{\Gamma, \texttt{true} \rightsquigarrow_b \mathit{true}}\ \text{(true)} \qquad \frac{}{\Gamma, \texttt{false} \rightsquigarrow_b \mathit{false}}\ \text{(false)} \qquad \frac{\Gamma, a_1 \rightsquigarrow_a v_1 \qquad \Gamma, a_2 \rightsquigarrow_a v_2}{\Gamma, (a_1 \text{ op } a_2) \rightsquigarrow_a (v_1 \text{ op}_{\mathbb{B}}\ v_2)}\ (\text{bops}_1)$$

$$\frac{\Gamma, b_1 \rightsquigarrow_b v_1 \qquad \Gamma, b_2 \rightsquigarrow_b v_2}{\Gamma, (b_1 \text{ op } b_2) \rightsquigarrow_b (v_1 \text{ op}_{\mathbb{B}}\ v_2)}\ (\text{bops}_2) \qquad \frac{\Gamma, b_1 \rightsquigarrow_b v_1}{\Gamma, (\sim b_1) \rightsquigarrow_b \sim (v_1)}\ (\text{bops}_3)$$

Implementation (OCaml)

```
let rec evalAexpr(a: aexpr) (m: state): aexpr =
  match a with
  | Aconst i        → Aconst i
  | Var s           → Aconst(lookup m s)
  | Plus(a1, a2)    → let ea1 = evalAexpr a1 m in
                      let ea2 = evalAexpr a2 m in
                      begin
                        match (ea1, ea2) with
                        | (Aconst v1, Aconst v2) → Aconst(v1+v2)
                        | (_, _)                 → Plus(ea1, ea2)
                      end
  | Mult(a1, a2)    → let ea1 = evalAexpr a1 m in
                      let ea2 = evalAexpr a2 m in
                      begin
                        match (ea1, ea2) with
                        | (Aconst v1, Aconst v2) → Aconst(v1*v2)
                        | (_, _)                 → Mult(ea1, ea2)
                      end

                        ...
```

Implementation (OCaml)

```
let rec evalBexpr(b: bexpr) (m: state): bexpr =
  match b with
   | Bconst b     → Bconst b
   | And(b1, b2) → let eb1 = evalBexpr b1 m in
                    let eb2 = evalBexpr b2 m in
                    begin
                      match (eb1, eb2) with
                      | (Bconst v1, Bconst v2) → Bconst(v1 && v2)
                      | (_, _)                 → And(eb1, eb2)
                    end
   | Eq(a1, a2) →  let ea1 = evalAexpr a1 m in
                    let ea2 = evalAexpr a2 m in
                    begin
                      match (ea1, ea2) with
                      | (Aconst v1, Aconst v2) → Bconst(v1 = v2)
                      | (_, _)                 → Eq(ea1, ea2)
                    end

                        ...
```

The IMP Language (Small-step Operational Semantics – Commands)

$\rightsquigarrow_c \quad : \text{ command} \rightarrow \Gamma \rightarrow \Gamma$

The IMP Language (Small-step Operational Semantics – Commands)

$\rightsquigarrow_c \quad : \text{ command} \rightarrow \Gamma \rightarrow \Gamma$

$$\frac{}{\Gamma, \texttt{skip} \rightsquigarrow_c \Gamma}\ (\text{skip})$$

$$\frac{}{\Gamma, (\texttt{int}\ \ x) \rightsquigarrow_c (x, 0) :: \Gamma}\ (\text{decl})$$

$$\frac{\Gamma, c_1 \rightsquigarrow_c \Gamma' \qquad \Gamma', c_2 \rightsquigarrow_c \Gamma''}{\Gamma, (c_1 ; c_2) \rightsquigarrow_c \Gamma''}\ (\text{seq})$$

$$\frac{\Gamma, a \rightsquigarrow_a v}{\Gamma, x := a \rightsquigarrow_c \Gamma[x \leftarrow v]}\ (\text{assign})$$

$$\frac{\Gamma, b \rightsquigarrow_b \mathit{true}}{\Gamma, \left(\texttt{if}(b)\texttt{then}\{c_1\}\texttt{else}\{c_2\}\right) \rightsquigarrow_c \Gamma, c_1}\ (\text{ite}_1)$$

$$\frac{\Gamma, b \rightsquigarrow_b \mathit{false}}{\Gamma, \left(\texttt{if}(b)\texttt{then}\{c_1\}\texttt{else}\{c_2\}\right) \rightsquigarrow_c \Gamma, c_2}\ (\text{ite}_2)$$

$$\frac{\Gamma, b \rightsquigarrow_b \mathit{true}}{\Gamma, \left(\texttt{while}(b)\{c\}\right) \rightsquigarrow_c \Gamma, \left(c ; \texttt{while}(b)\{c\}\right)}\ (\text{loop}_1)$$

$$\frac{\Gamma, b \rightsquigarrow_b \mathit{false}}{\Gamma, \left(\texttt{while}(b)\{c\}\right) \rightsquigarrow_c \Gamma}\ (\text{loop}_2)$$

Implementation (OCaml)

```
let rec evalCmd(c: cmd) (m: state): state =
  match c with
  | Skip             → m
  | Declare s        → update m s 0
  | Assign(s, a)     → let ea = evalAexpr a m in
                       begin
                         match ea with
                           | Aconst v → update m s v
                           | _        → failwith "assignment_error"
                       end
  | Sequence(c1, c2) → let m' = evalCmd c1 m  in evalCmd c2 m'
  | Ite(b, c1, c2)   → let eb = evalBexpr b m in
                       begin
                         match eb with
                         | Bconst v → if v then evalCmd c1 m else evalCmd c2 m
                         | _        → failwith "ite_error"
                       end
  | While(b, c1)     → let eb = evalBexpr b m in
                       begin
                        match eb with
                        | Bconst v → if v then evalCmd(Sequence(c1, (While(b, c1)))) m else m
                        | _        → failwith "while_error"
                       end
```

Scaling Up . . .

- operational semantics (and similarly styled typing rules) can handle full languages

Scaling Up . . .

- operational semantics (and similarly styled typing rules) can handle full languages
  - with records, recursive variant types, objects, first-class functions, and more

Scaling Up . . .

- operational semantics (and similarly styled typing rules) can handle full languages
  - with records, recursive variant types, objects, first-class functions, and more
- provides a concise notation for explaining what a language does by clarifying:

Scaling Up . . .

- operational semantics (and similarly styled typing rules) can handle full languages
  - with records, recursive variant types, objects, first-class functions, and more
- provides a concise notation for explaining what a language does by clarifying:
  - evaluation order

Scaling Up . . .

- operational semantics (and similarly styled typing rules) can handle full languages
  - with records, recursive variant types, objects, first-class functions, and more
- provides a concise notation for explaining what a language does by clarifying:
  - evaluation order
  - call-by-value vs. call-by-name

Scaling Up . . .

- operational semantics (and similarly styled typing rules) can handle full languages
  - with records, recursive variant types, objects, first-class functions, and more
- provides a concise notation for explaining what a language does by clarifying:
  - evaluation order
  - call-by-value vs. call-by-name
  - . . .

Thanks! & Questions?