The Constructive Semantics

- Logical correctness is not in accordance with the intention of the language, ie with its intuitive semantics and with the intended sequential character of test statements.
- Example:

  ```
  module P10:
  present O then nothing end; emit O
  ```

  is logically correct, but the information that O is present flows backwards across the sequencing operator ; contradicting the basic intuition about sequential execution.
- Aside from the explicit concurrency || all Esterel statements are sequential.

The Constructive Semantics

- Idea: do not check assumptions about signal statuses, but propagate facts about control flow and signal statuses. Self-justification is replaced by fact-to-fact propagation.
- Accounts for programmer's natural way of thinking: in terms of cause and effect.
- Three-valued logic for signals: present, absent, unknown.
- In each instant the statuses of the input signals are given by the environment and the statuses of the other signals are initially set to unknown.

The Constructive Semantics

- Three equivalent presentations:
  - Constructive behavioral semantics
    - Derived from the logical behavioral semantics
    - Constructive restrictions are added to the logical coherence rule
  - Constructive operational semantics
    - Based on term rewriting rules defining microstep sequences
    - Simplest way of defining an efficient interpreter
  - Circuit semantics
    - Translation of program into constructive circuits
    - Core of the Esterel v5 compiler.

Constructive Behavioral Semantics

- Logical coherence semantics augmented by reasoning about what a program must or cannot do, both predicates being disjoint and defined in a constructive way.

- The must predicate determines which signals are present and which statements are executed.
- The cannot predicate determines when signals are absent and it serves in pruning out false execution paths.

- A program is accepted as constructive if and only if fact propagation using the must and cannot predicates suffices in establishing presence or absence of all signals.

Constructive Behavioral Semantics

- Logical Coherence Law:
  - A signal S is present in an instant iff an emit S statement is executed in this instant.
- Constructive Coherence Law:
  - A signal S is present iff an emit S statement must be executed.
  - A signal S is absent iff an emit S statement cannot be executed.

Constructive Behavioral Semantics

- A signal can have three statuses:
  - +: known to be present
  - –: known to be absent
  - ⊥: yet unknown
- must and cannot predicates are defined by structural induction on statements.

- *p* ; *q*
  - Must (resp. can) execute *q* if *p* must (resp. can) terminate
- present *S* then *p* else *q* end
  - *S* known to be present -> Test behaves as *p*
  - *S* known to be absent -> Test behaves as *q*
  - *S* yet unknown -> Test can do whatever *p* or *q* can do; there is nothing the test must do.

Example 1

```
module P1:
input I;
output O;
signal S1, S2 in
  present I then emit S1 end (i1)
  ||
  present S1 else emit S2 end (i2)
  ||
  present S2 then emit O end (i3)
end signal
end module
```

- If I is present:
  - i1 must take its then branch, emit S1 and terminate → S1 present
  - i2 must take its (empty) then branch and cannot take its else branch → emit S2 cannot be executed, S2 cannot be emitted → S2 absent
  - i3 cannot take its then branch → O cannot be emitted and is absent.
- If I is absent:
  - i1 cannot take its then branch → emit S1 cannot be executed → S1 absent
  - i2 must take its then branch → emit S2 must be executed → S2 present.
  - i3 must take its then branch → emit O must be executed → O present.

Example 2 – Part 1

```
module P2:
output O;
signal S in
  emit S;
  present O then
    present S then
      pause
    end;
    emit O
  end
end signal
```

- Analyze what signal S must do with status ⊥ for O.
  - Analyze body with status ⊥ for O and S.
  - S must be emitted.
  - Thus: redo the analysis with status ⊥ for O and + for S.
  - Status of O is unknown: there is nothing that the present statement must do. Progress can only be made by analyzing what we cannot do in the branches of the test.
  - The *then* branch contains a present S test. Since S is known to be present, we cannot take the implicit *else* branch. Since the *then* branch is a pause statement it cannot terminate. Therefore the emit O statement cannot be executed and O cannot be emitted.
  - As a consequence O must be set absent and the analysis must be redone with status – for O.

Example 2 – Part 2

```
module P2:
output O;
signal S in
  emit S;
  present O then
    present S then
      pause
    end;
    emit O
  end
end signal
```

- Analyze what signal S must do with status – for O.
  - The implicit else branch of the present O test that terminates execution must be taken.
  - The program is constructive since we have fully determined the signal statuses.

Constructive Behavioral Semantics

- signal *S* in *p* end
  - Can: recursively analyze *p* with status ⊥ for *S*
  - Must:
    - Assume we already know that we must execute the declaration in some signal context *E*
    - Must compute final status of *S* to determine signal context of *p*
    - First analyze *p* in *E* augmented by setting the unknown status ⊥ for *S*
    - If *S* must be emitted:
      - propagate this information by reanalyzing *p* in *E* with *S* present
      - This may generate more information about the other signals
    - If S cannot be emitted:
      - reanalyze *p* in *E* with *S* absent

Constructive Behavioral Semantics – Formal Definition

- Let S be a set of signals.

  An event E is a mapping $E: S \rightarrow B_{\perp} = \{+, -, \perp\}$ which assigns a status from $B_{\perp}$ to all signals in S.

- Notation:
  - $s^{+}: E(s) = +$
  - $s^{-}: E(s) = -$
  - $E \subseteq E'$: s^{+} in $E \Rightarrow s^{+}$ in E'

- Singleton event $\{s^{+}\}$:

  $\{s^{+}\}(s) = +$ and $\{s^{+}\}(s') = -$ forall $s' \neq s$

- Let an event E for a set S be given, a signal s possibly not in S and a status b in $B_{\perp}$. Then $E * s^{b}$ is an event for the set $S \cup \{s\}$ where

  $$E*s^{b}(s) = b \text{ and } E*s^{b}(s') = E(s') \; \forall \; s' \neq s.$$

Constructive Behavioral Semantics – Formal Definition

- The statements nothing, pause and exit are represented by completion codes *k* >= 0:
  - nothing is encoded by 0
  - pause is encoded by 1
  - exit T is encoded by 2, if the directly enclosing trap declaration is that of T and *n* +2 if *n* trap declarations have to be traversed before reaching that of T.
- To handle trap propagation we define two operators

$$\downarrow k = \begin{cases} 0, & \text{if } k = 0 \text{ or } k = 2 \\ 1, & \text{if } k = 1 \\ k - 1, & \text{if } k > 2 \end{cases}$$

$$\uparrow k = \begin{cases} k, & \text{if } k = 0 \text{ or } k = 2 \\ k + 1, & \text{if } k > 1 \end{cases}$$

Constructive Behavioral Semantics – Formal Definition

- Given a program P with body p and an input event I. A reaction of the program is given by a behavioral transition of the form

  $$P \xrightarrow[I]{O} P'$$

  where O is an output event and the resulting program P' is the new state reached by P after the reaction. P' is called the derivative of P by the reaction.
- The statement transition relation has the form

  $$p \xrightarrow[E]{E',k} p'$$

  where
  - E is an event that defines the status of all signals in the scope of p
  - E' is an event composed of all signals emitted by p in the reaction, k is the completion code returned.

The statement p' is called the derivative of p by the reaction.

Constructive Behavioral Semantics – Formal Definition

$$P \xrightarrow[I]{O} P' \quad \Leftrightarrow \quad p \xrightarrow[I \cup O]{O,k} p' \ \text{ for some } k$$

$$Max(K,L) = \begin{cases} \emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\ \{\max\{k,l\}\}, & \text{for } k \in K, l \in L \end{cases}$$

Constructive Behavioral Semantics – Formal Definition

- The Must function determines what must be done in a reaction
.
$$P \xrightarrow[I]{O} P'$$

  $$\mathrm{Must}(p, E) = \langle S, K \rangle$$

  where
  - E is an event,
  - S is the set of signals that p must emit
  - K is the set of completion codes that p must return.

- We write
$\mathrm{Must}(p, E) = \langle S, K \rangle =: \langle \mathrm{Must}_s(p,E), \mathrm{Must}_k(p,E) \rangle$.

Constructive Behavioral Semantics – Formal Definition

- The function $\mathrm{Cannot}^m(p,E)$ is used to prune out false paths.

  $\mathrm{Cannot}^m(p,E) = \langle \mathrm{Cannot}_s^m(p,E),\ \mathrm{Cannot}_k^m(p,E) \rangle = \langle S,K \rangle$

  - S is the set of signals that p cannot emit
  - K is the set of completion codes that p cannot exit with when the input event is E.

- $m \in \{+, \bot\}$ indicates whether it is known that the statement p must be executed in the event E. The case $m = -$ will never occur since Cannot will only be called for potentially executable statements.

- In the following, we will use $\mathrm{Can}^m(p,E)$ since it is easier to be defined formally; from this, $\mathrm{Cannot}^m(p,E)$ can be determined by componentwise complementation.

Constructive Behavioral Semantics – Formal Definition

- Must and Can^m are defined by structural induction over the kernel statements.

$$Must(k,E) = Can^m(k,E) = \langle \emptyset, \{k\} \rangle$$

$$Must(emit\ S,E) = Can^m(emit\ S,E) = \langle \{s\}, \{0\} \rangle$$

$$Must(present\ s\ then\ p\ else\ q\ end,E) = \begin{cases} Must(p,E), & if\ s^{+} \in E \\ Must(q,E), & if\ s^{-} \in E \\ \langle \emptyset, \emptyset \rangle, & if\ s^{\perp} \in E \end{cases}$$

$$Can^m(present\ s\ then\ p\ else\ q\ end,E) = \begin{cases} Can^m(p,E), & if\ s^{+} \in E \\ Can^m(q,E), & if\ s^{-} \in E \\ Can^{\perp}(p,E) \cup Can^{\perp}(q,E), & if\ s^{\perp} \in E \end{cases}$$

$$Must(suspend\ \ p\ when\ s,E) = Must(p,E)$$
$$Can^m(suspend\ \ p\ when\ s,E) = Can^m(p,E)$$

Constructive Behavioral Semantics – Formal Definition

$$Must(p;q,E)=\begin{cases} Must(p,E), & if\ \ Must_k(p,E)\neq\{0\} \\ \left\langle Must_s(p,E)\cup Must_s(q,E), Must_k(q,E)\right\rangle, & if\ \ Must_k(p,E)=\{0\} \end{cases}$$

We analyze q only if p must terminate in which case the completion code 0 of p is discarded.

Constructive Behavioral Semantics – Formal Definition

$$Can^{m}(p;q,E)=\begin{cases} Can^{m}(p,E), & \\ & \text{if } 0\notin Can_{k}^{m}(p,E) \\ \left\langle Can_{s}^{m}(p,E)\cup Can_{s}^{m'}(q,E), Can_{k}^{m}(\mathrm{p,E})\setminus\{0\}\cup Can_{k}^{m'}(q,E)\right\rangle & \\ & \text{if } 0\in Can_{k}^{m}(p,E) \text{ with } m'=+ \text{ if } m=+ \\ & \text{and } 0\in Must_{k}(p,E) \\ & \text{or if } 0\in Can_{k}^{m}(p,E) \text{ with } m'=\perp \text{ otherwise} \end{cases}$$

We analyze q with argument $m'=+$ if $m=+$ and if p must terminate, with argument $m'=\perp$ otherwise.

Constructive Behavioral Semantics – Formal Definition

$$Must(loop\ p\ end, E) = Must(p, E)$$

$$Can^m(loop\ p\ end, E) = Can^m(p, E)$$

$$Must(p \| q, E) = \left\langle Must_s(p, E) \cup Must_s(q, E), Max(Must_k(p, E), Must_k(q, E)\right\rangle$$

$$Can^m(p \| q, E) = \left\langle Can_s^m(p, E) \cup Can_s^m(q, E), Max(Can_k^m(p, E), Can_k^m(q, E)\right\rangle$$

- The Max operation e.g. ensures that || cannot terminate if one of its branches cannot do so.

Constructive Behavioral Semantics – Formal Definition

$$Must(trap\,T\ in\ p\ end, E) = Must(\{\uparrow p\})$$

$$Must(\{q\}, E) = \left\langle Must_s(q,E), \downarrow Must_k(q,E) \right\rangle$$

$$Must(\uparrow q, E) = \left\langle Must_s(q,E), \uparrow Must_k(q,E) \right\rangle$$

$$Can^m(trap\,T\ in\ p\ end, E) = Can^m(\{\uparrow p\})$$

$$Can^m(\{q\}, E) = \left\langle Can_s^m(q,E), \downarrow Can_k^m(q,E) \right\rangle$$

$$Can^m(\uparrow q, E) = \left\langle Can_s^m(q,E), \uparrow Can_k^m(q,E) \right\rangle$$

Constructive Behavioral Semantics – Formal Definition

$$Must(signal\ s\ in\ p,E) = \begin{cases} Must(p,E*s^{+})\setminus\{s\}, & if\ \ s \in Must_s(p,E*s^{\perp}) \\ Must(p,E*s^{-})\setminus\{s\}, & if\ \ s \notin Can_s^{+}(p,E*s^{\perp}) \\ Must(p,E*s^{\perp})\setminus\{s\}, & otherwise \end{cases}$$

$$Can^{m}(signal\ s\ in\ p,E) = \begin{cases} Can^{+}(p,E*s^{+})\setminus\{s\}, & if\ \ m=+\ \ and\ \ s \in Must_s(p,E*s^{\perp}) \\ Can^{m}(p,E*s^{-})\setminus\{s\}, & if\ \ s \notin Can_s^{+}(p,E*s^{\perp}) \\ Can_s^{m}(p,E*s^{\perp})\setminus\{s\}, & otherwise \end{cases}$$

- We first analyze the body *p* with status ⊥ for *s* with the same *m* argument.
- If *m*=+ and we find that the signal must be emitted we reanalyze *p* with status + for *s*.
- For both *m*=+ and *m*= ⊥ if the signal cannot be emitted we reanalyze *p* with status – and with the same *m*.
- Otherwise we return the result of the analysis of *p* with status ⊥ for *s*.
- Note that the signal status can be set to + only if *m*=+. This is necessary to avoid speculative computations.

Constructive Behavioral Semantics

- The constructiveness analysis involves many recomputations: Once a signal status has been set, the body of its declaration (the whole program for an output) has to be reanalyzed, this way re-establishing many facts that are already known.
- The goal of the operational and circuit semantics is to avoid recomputing known facts.

Example

```
module P4:
input I;
output O;
signal S1, S2 in
  present I then emit S1 end
||
  present S1 then emit S2 end
||
  present S2 then emit O end
end module
```

accepted by constructiveness

```
module P3:
input I;
output O1,O2;
present I then
  present O2 then emit O1 end
else
  present O1 then emit O2 end
end present
end module
```

rejected by acyclicity test
reactive and deterministic
accepted by constructiveness

Examples

```
module P1:
output O;
  present O
    else emit O
  end present
end module
```

rejected by constructiveness

```
module P2:
output O;
  present O
    then emit O
  end present
end module
```

rejected by constructiveness

Examples

```
module Px:
output O;
  present O then emit O else emit O
end module
```

logically correct by self justification
rejected by constructiveness

Advanced Constructiveness

- Preemption statements behave as tests for the guard in each instant where the guard is active. Their constructiveness test is straightforward.

```
module Py:
output O;
abort
  sustain O
when O
```

non-constructive in the first instant
non-constructive (non reactive) in later instants

Advanced Constructiveness

- Preemption statements (abort) behave as tests for the guard in each instant where the guard is active. Their constructiveness test is straightforward.
- Signal expressions:
  - not e: straightforward
  - e1 or e2: evaluates to true as soon as one of e1 or e2 evaluates to true, even if the other one is still unknown.
  - e1 and e2: analogous
- The computation of values of valued signals cannot be lazy since the value is known only when all emitters are either executed or discarded (due to signal combination).
  A statement such as emit S(2) is handled as
  emit S; ?S:=2; by the constructiveness test.

Advanced Constructiveness

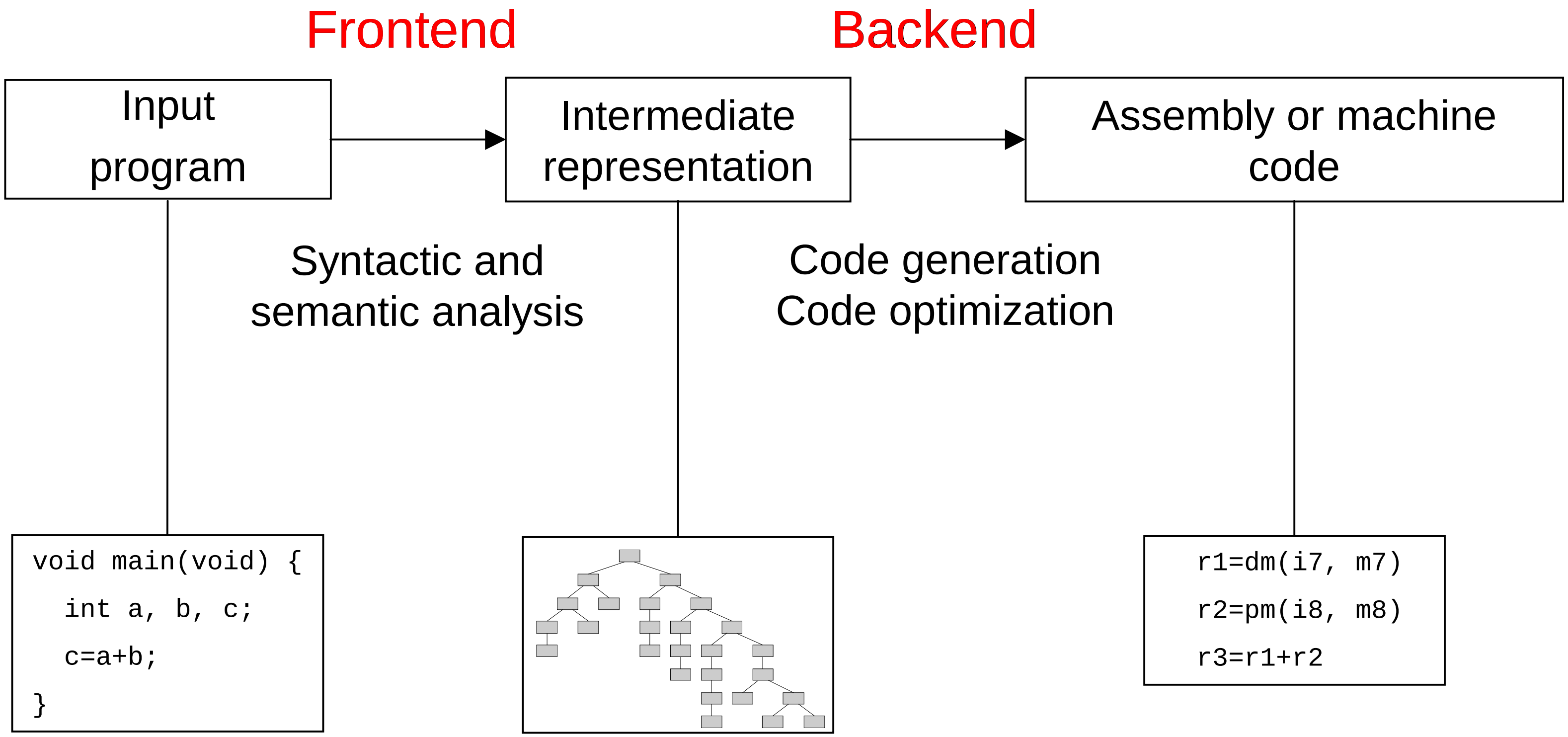
- Signal expressions:
  - not e: straightforward
  - e1 or e2: evaluates to true as soon as one of e1 or e2 evaluates to true, even if the other one is still unknown.
  - e1 and e2: analogous
- The computation of values of valued signals cannot be lazy since the value is known only when all emitters are either executed or discarded (due to signal combination).
  A statement such as emit S(2) is handled as
  emit S; ?S:=2; by the constructiveness test.

Example

```
signal S1, S2 in
  present I then emit S1 else emit S2
||
  present S1 then
    call P1()();
    emit S2
  end present
||
  present S2 then
    call P2()();
    emit S1
  end present
end signal
```

constructive

Compiler Structure



Frontend

Backend

Input program

Intermediate representation

Assembly or machine code

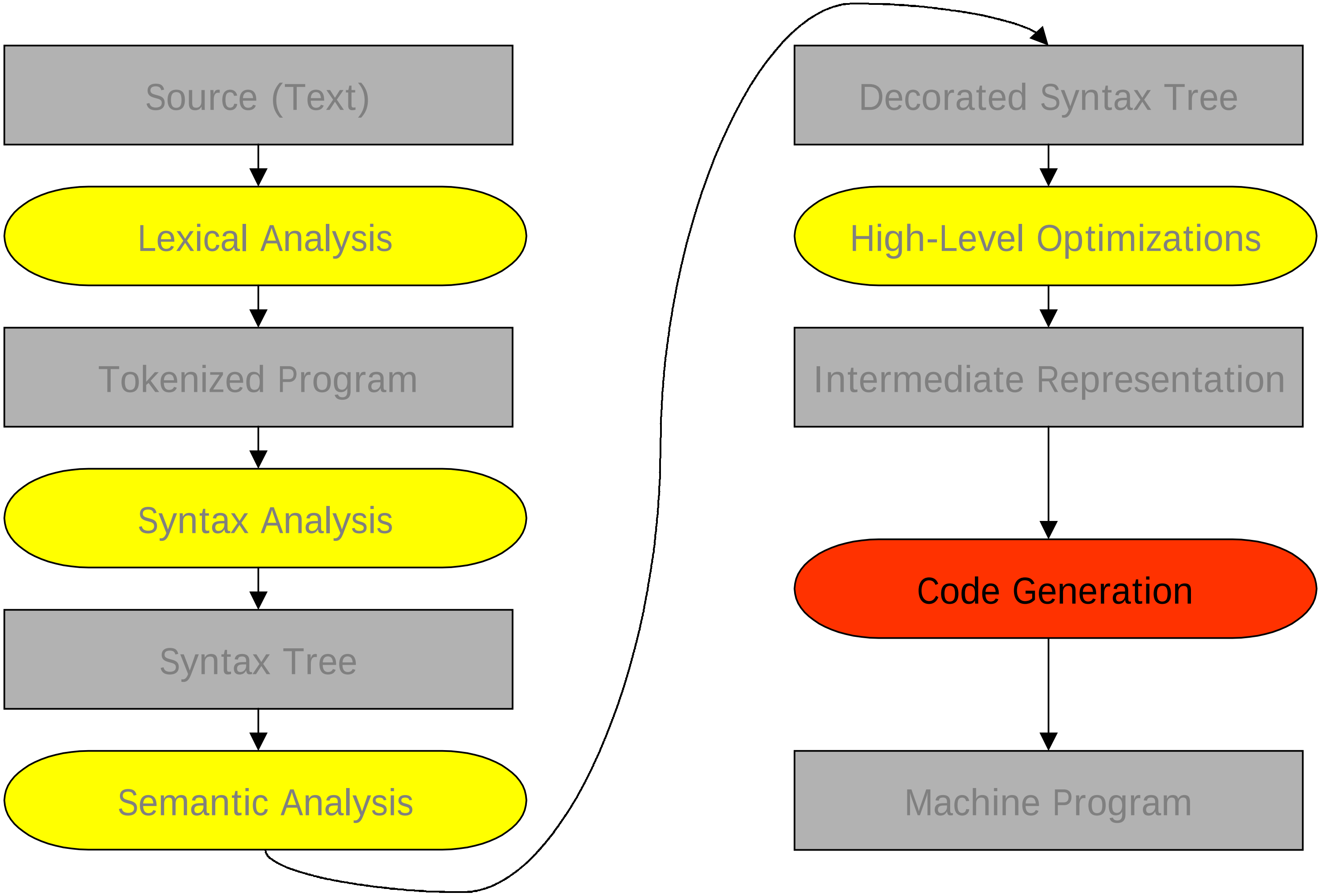
Syntactic and semantic analysis

Code generation
Code optimization

```
void main(void) {
  int a, b, c;
  c=a+b;
}
```

```
r1=dm(i7, m7)
r2=pm(i8, m8)
r3=r1+r2
```

Detailed Compiler Structure



Source (Text)

Lexical Analysis

Tokenized Program

Syntax Analysis

Syntax Tree

Semantic Analysis

Decorated Syntax Tree

High-Level Optimizations

Intermediate Representation

Code Generation

Machine Program