Compiler Construction

Lecture 17: Semantic Analysis V (Attribute Evaluation)/ Code Generation I (Foundations)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc09/

Winter semester 2009/10

Outline

Attribute Evaluation Methods

Given:
- (strongly) noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
- syntax tree t of G
- valuation $v : Syn_\Sigma \to V$ where $Syn_\Sigma := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) solution $v : Var_t \to V$

Methods:

1. Topological sorting of D_t:
   1. start with attribute variables which depend at most on synthesized attributes of terminals (Syn_Σ)
   2. proceed by successive substitution
2. Recursive functions (for strongly noncircular AGs):
   1. for every $A \in N$ and $\alpha \in \text{syn}(A)$, define evaluation function $g_{A,\alpha}$ with the following parameters:
      - the node of t where α has to be evaluated and
      - all inherited attributes of A on which α (potentially) depends
   2. for every $\alpha \in \text{syn}(S)$, evaluate $g_{S,\alpha}(k_0)$ where k_0 denotes the root of t
3. Special cases: S-attributed grammars (`yacc`), L-attributed grammars

Attribute Evaluation by Recursive Functions

Restriction: only for strongly noncircular attribute grammars

Principle:

1. for every $A \in N$ and $\alpha \in \text{syn}(A)$, define evaluation function $g_{A,\alpha}$ with the following parameters:
   - the node of t where α has to be evaluated (which is labelled by A) and
   - all inherited attributes of A on which α (potentially) depends (that is, $\{\beta \in \text{inh}(A) \mid (\beta, \alpha) \in IS'(A)\}$)
2. given a syntax tree t with root k_0, evaluate $g_{S,\alpha}(k_0)$ for every $\alpha \in \text{syn}(S)$

Result: evaluates synthesized attribute variables at root of t and all attribute variables on which they actually depend (according to E_t)

Definition of Evaluation Functions I

For every $A \in N$ and $\alpha \in \text{syn}(A)$, let

- $IS'(A) \subseteq \text{inh}(A) \times \text{syn}(A)$ as computed by strong circularity test (Algorithm 16.2)
- $\text{inh}(A, \alpha) := \{\beta \in \text{inh}(A) \mid (\beta, \alpha) \in IS'(A)\}$
- $A \to \delta_1 \mid \ldots \mid \delta_m$ all A-productions in P

Then $g_{A,\alpha}$ is given by

$$g_{A,\alpha}(k_0, \text{inh}(A, \alpha)) := \textbf{case} \text{ production applied at } k_0 \textbf{ of}$$
$$\vdots$$
$$A \to \delta_j : \text{eval}(\alpha.0)$$
$$\vdots$$
$$\textbf{end}$$

with

$$\text{eval}(\gamma.i) := \begin{cases} \gamma & \text{if } \gamma \in Inh, i = 0 \\ f(\text{eval}(\gamma_1.i_1), \ldots, \text{eval}(\gamma_n.i_n)) & \text{if } \gamma.i \in In_{A \to \delta_j}, \gamma.i = f(\gamma_1.i_1, \ldots, \gamma_n.i_n) \in E_{A \to \delta_j} \\ g_{Y_i,\gamma}(k_i, \text{eval}(\beta_1.i), \ldots, \text{eval}(\beta_l.i)) & \text{if } \gamma \in Syn, i > 0, Y_i \in N, \text{inh}(Y_i, \gamma) = \{\beta_1, \ldots, \beta_l\} \\ v(\gamma.i) & \text{if } \gamma \in Syn, i > 0, Y_i \in \Sigma \end{cases}$$

where $\delta_j = Y_1 \ldots Y_r$, and where k_i denotes the ith successor of k_0

Definition of Evaluation Functions II

Example (cf. Example 14.1)

G'_B:

$S \to L$	$v.0 = v.1$
	$p.1 = 0$
$S \to L\,.\,L$	$v.0 = v.1 + v.3$
	$p.1 = 0$
	$p.3 = -l.3$
$L \to B$	$v.0 = v.1$
	$l.0 = 1$
	$p.1 = p.0$
$L \to LB$	$v.0 = v.1 + v.2$
	$l.0 = l.1 + 1$
	$p.1 = p.0 + 1$
	$p.2 = p.0$
$B \to \mathtt{0}$	$v.0 = 0$
$B \to \mathtt{1}$	$v.0 = 2^{p.0}$

$A \in N$	S	L	B
$IS'(A)$	$\emptyset$	$\{(p, v)\}$	$\{(p, v)\}$

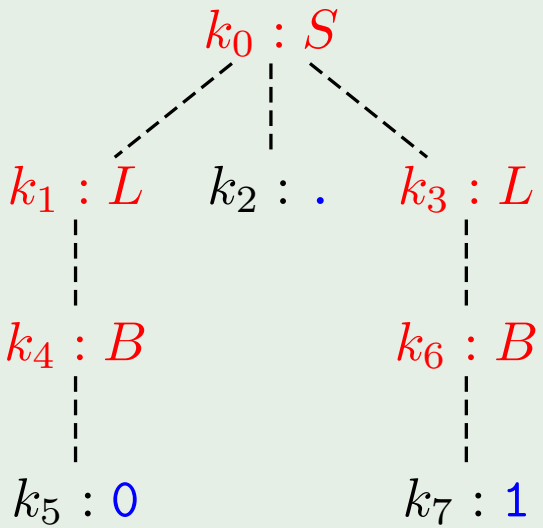
$$
\begin{aligned}
g_{S,v}(k_0) = \ & \textbf{case}\ \text{production}(k_0)\ \textbf{of} \\
& \quad S \to L : g_{L,v}(k_1, 0) \\
& \quad S \to L\,.\,L : g_{L,v}(k_1, 0) + \\
& \qquad g_{L,v}(k_3, -g_{L,l}(k_3)) \\
& \textbf{end} \\
g_{L,v}(k_0, p) = \ & \textbf{case}\ \text{production}(k_0)\ \textbf{of} \\
& \quad L \to B : g_{B,v}(k_1, p) \\
& \quad L \to LB : g_{L,v}(k_1, p + 1) \\
& \qquad + g_{B,v}(k_2, p) \\
& \textbf{end} \\
g_{L,l}(k_0) = \ & \textbf{case}\ \text{production}(k_0)\ \textbf{of} \\
& \quad L \to B : 1 \\
& \quad L \to LB : g_{L,l}(k_1) + 1 \\
& \textbf{end} \\
g_{B,v}(k_0, p) = \ & \textbf{case}\ \text{production}(k_0)\ \textbf{of} \\
& \quad B \to \mathtt{0} : 0 \\
& \quad B \to \mathtt{1} : 2^p \\
& \textbf{end}
\end{aligned}
$$

Example Evaluation

Example (continued)

$$
\begin{aligned}
g_{S,v}(k_0) = \ & \textbf{case } \text{production}(k_0) \textbf{ of} \\
& \quad S \to L : g_{L,v}(k_1, 0) \\
& \quad S \to L\,.\,L : g_{L,v}(k_1, 0) + \\
& \qquad g_{L,v}(k_3, -g_{L,l}(k_3)) \\
& \textbf{end} \\
g_{L,v}(k_0, p) = \ & \textbf{case } \text{production}(k_0) \textbf{ of} \\
& \quad L \to B : g_{B,v}(k_1, p) \\
& \quad L \to LB : g_{L,v}(k_1, p+1) \\
& \qquad + g_{B,v}(k_2, p) \\
& \textbf{end} \\
g_{L,l}(k_0) = \ & \textbf{case } \text{production}(k_0) \textbf{ of} \\
& \quad L \to B : 1 \\
& \quad L \to LB : g_{L,l}(k_1) + 1 \\
& \textbf{end} \\
g_{B,v}(k_0, p) = \ & \textbf{case } \text{production}(k_0) \textbf{ of} \\
& \quad B \to \texttt{0} : 0 \\
& \quad B \to \texttt{1} : 2^p \\
& \textbf{end}
\end{aligned}
$$

Syntax tree t:



$$
\begin{aligned}
& g_{S,v}(k_0) \\
&= g_{L,v}(k_1, 0) + g_{L,v}(k_3, -g_{L,l}(k_3)) \\
&= g_{B,v}(k_4, 0) + g_{L,v}(k_3, -g_{L,l}(k_3)) \\
&= 0 + g_{L,v}(k_3, -g_{L,l}(k_3)) \\
&= 0 + g_{B,v}(k_6, -g_{L,l}(k_3)) \\
&= 0 + 2^{-g_{L,l}(k_3)} \\
&= 0 + 2^{-1} \\
&= 0.5
\end{aligned}
$$

Outline

Why Strong Noncircularity?

If the attribute grammar is not strongly noncircular, then the construction of the evaluation functions fails.

Example 17.1 (cf. Example 16.3)

$$
\begin{array}{lll}
S \to A & \alpha.0 = \alpha_2.1 \\
 & \beta_1.1 = \alpha_1.1 \\
 & \beta_2.1 = \alpha_2.1 \\
A \to \texttt{a} & \alpha_1.0 = \beta_2.0 \\
 & \alpha_2.0 = 2 \\
A \to \texttt{b} & \alpha_1.0 = 1 \\
 & \alpha_2.0 = \beta_1.0
\end{array}
$$

From Example 16.3:

$IS'(A) = \{(\beta_2, \alpha_1), (\beta_1, \alpha_2)\}$

Definition of $g_{S,\alpha}$:

$$
\begin{aligned}
& g_{S,\alpha}(k_0) \\
&= \mathrm{eval}(\alpha.0) \\
&= \mathrm{eval}(\alpha_2.1) \\
&= g_{A,\alpha_2}(k_1, \mathrm{eval}(\beta_1.1)) \\
&= g_{A,\alpha_2}(k_1, \mathrm{eval}(\alpha_1.1)) \\
&= g_{A,\alpha_2}(k_1, g_{A,\alpha_1}(k_1, \mathrm{eval}(\beta_2.1))) \\
&= g_{A,\alpha_2}(k_1, g_{A,\alpha_1}(k_1, \mathrm{eval}(\alpha_2.1))
\end{aligned}
$$

$\Longrightarrow$ does not terminate!

Outline

L-Attributed Grammars I

In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed to run from left to right.

Definition 17.2 (L-attributed grammar)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ such that, for every $\pi \in P$ and $\beta.i = f(\ldots, \alpha.j, \ldots) \in E_\pi$ with $\beta \in Inh$ and $\alpha \in Syn$, $j < i$. Then $\mathfrak{A}$ is called an L-attributed grammar (notation: $\mathfrak{A} \in LAG$).

Corollary 17.3

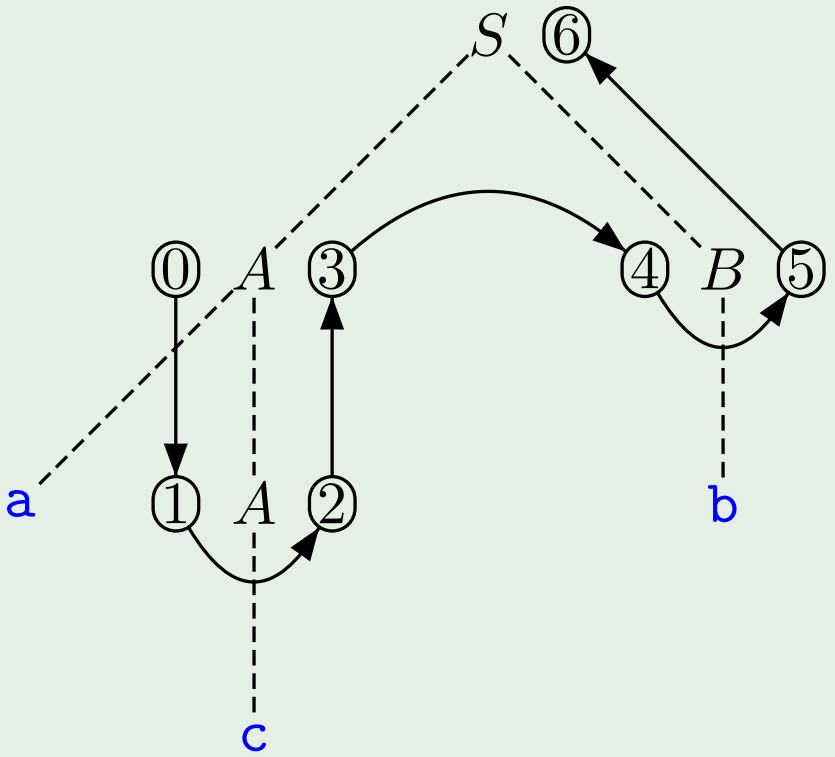
Every $\mathfrak{A} \in LAG$ is noncircular.

L-Attributed Grammars II

Example 17.4

L-attributed grammar:

$S \to AB$	$i.1$	$= 0$
	$i.2$	$= s.1 + 1$
	$s.0$	$= s.2 + 1$
$A \to \mathtt{a}A$	$i.2$	$= i.0 + 1$
	$s.0$	$= s.2 + 1$
$A \to \mathtt{c}$	$s.0$	$= i.0 + 1$
$B \to \mathtt{b}$	$s.0$	$= i.0 + 1$



Evaluation of L-Attributed Grammars

Observation 1: the syntax tree of an L-attributed grammar can be attributed by a depth-first, left-to-right tree traversal with two visits to each node

1. top-down: evaluation of inherited attributes
2. bottom-up: evaluation of synthesized attributes

Observation 2: visit sequence fits nicely with parsing

1. top-down: expansion steps
2. bottom-up: reduction steps

Idea: extend LL parsing to support reduction steps, and integrate attribute evaluation $\Longrightarrow$

- use recursive-descent parser
- add variables and operations for attribute evaluation

Recursive-Descent Parsing and Evaluation I

Ingredients:
- variable `token` for current token
- function `next()` for invoking the scanner
- procedure `print(i)` for displaying the leftmost analysis (or errors)

Method: to every $A \in N$ we assign a procedure

`A(in:` inh(A), `out:` syn(A)`)`

which
- declares local variables for synthesized attributes on right-hand sides,
- tests `token` with regard to the lookahead sets of the A-productions,
- prints the corresponding rule number and
- evaluates the corresponding right-hand side as follows:
  - for $a \in \Sigma$: check `token`; call `next()`
  - for $A \in N$: call `A` with appropriate parameters

Recursive- Parsing II

Example 17.5 (Arithmetic expressions; cf. Example 8.10)

```
proc main();
  token := next(); S()
proc S();    (* S → A B *)
  if token in {'a','c'} then
    print(1); A(); B()
  else print(error); stop fi
proc A();  (* A → a A | c *)
  if token = 'a' then
    print(2); token := next(); A()
  elsif token = 'c' then
    print(3); token := next()
  else print(error); stop fi
proc B();    (* B → b *)
  if token = 'b' then
    print(4); token := next()
  else print(error); stop fi
```

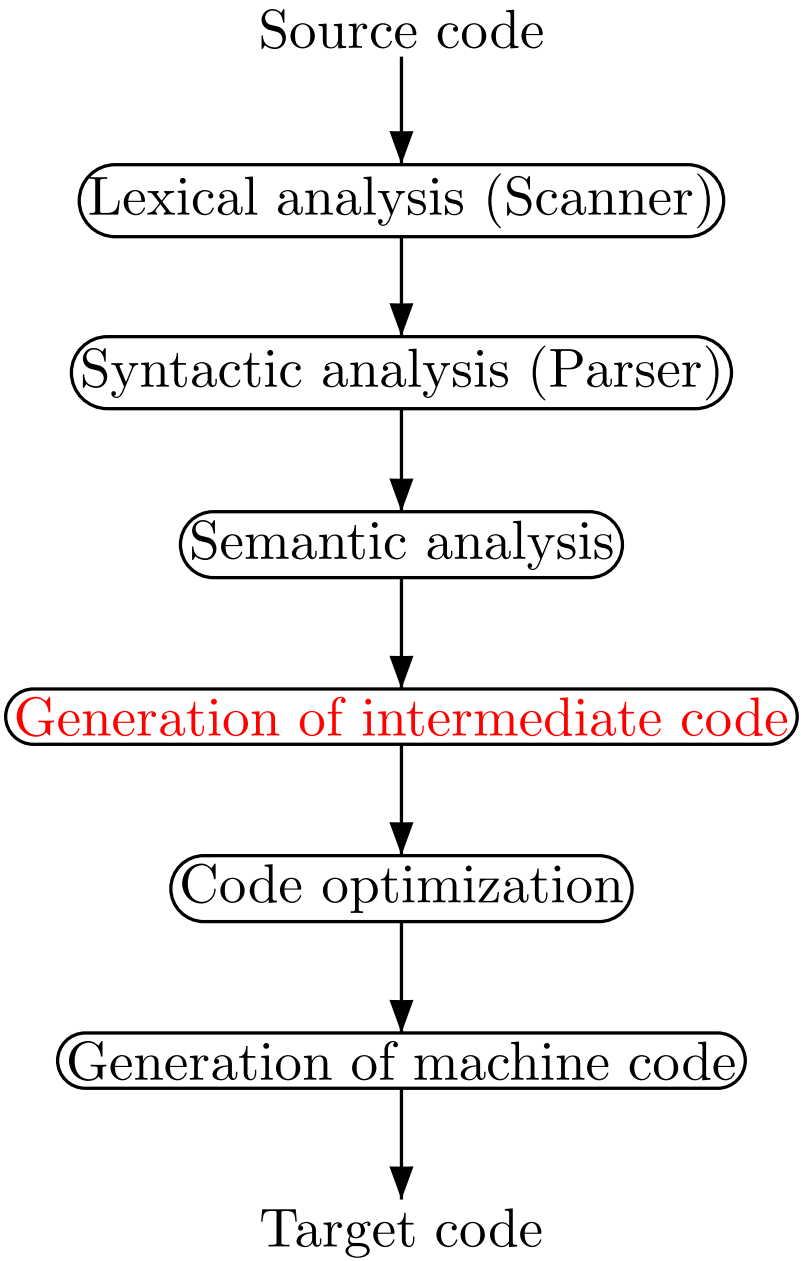
Recursive-Descent Parsing and Evaluation II

Example 17.6 (Arithmetic expressions; cf. Example 17.4)

```
proc main(); var s;
  token := next(); S(s); print(s)
proc S(out s0); var s1,s2;    (* S → A B *)
  if token in {'a','c'} then
    print(1); A(0,s1); B(s1 + 1,s2); s0 := s2 + 1
  else print(error); stop fi
proc A(in i0,out s0); var s2;    (* A → a A | c *)
  if token = 'a' then
    print(2); token := next(); A(i0 + 1,s2); s0 := s2 + 1
  elsif token = 'c' then
    print(3); token := next(); s0 := i0 + 1
  else print(error); stop fi
proc B(in i0,out s0);    (* B → b *)
  if token = 'b' then
    print(4); token := next(); s0 := i0 + 1
  else print(error); stop fi
```

Outline

Conceptual Structure of a Compiler



Modularization of Code Generation I

Splitting of code generation for programming language PL:

$$\text{PL} \xrightarrow{\text{trans}} \text{IC} \xrightarrow{\text{code}} \text{MC}$$

Frontend: trans generates machine-independent intermediate code (IC) for abstract (stack) machine

Backend: code generates actual machine code (MC)

Advantages: IC machine independent $\Longrightarrow$

Portability: much easier to write IC compiler/interpreter for a new machine (as opposed to rewriting the whole compiler)

Fast compiler implementation: generating IC much easier than generating MC

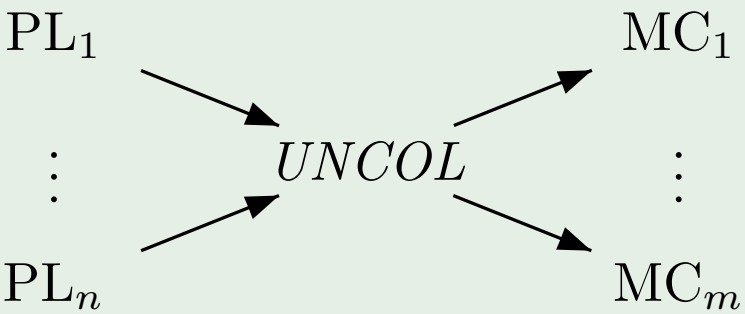
Code size: IC programs usually smaller than corresponding MC programs

Code optimization: division into machine-independent and machine-dependent parts

Modularization of Code Generation II

Example 17.7

1. UNiversal Computer-Oriented Language (UNCOL; $\approx$ 1960; `http://en.wikipedia.org/wiki/UNCOL`):
   universal intermediate language for compilers (never fully specified or implemented; too ambitious)



only $n + m$ translations (in place of $n \cdot m$)

2. Pascal's pseudocode (P-code; $\approx$ 1975; `http://en.wikipedia.org/wiki/P-Code_machine`)
3. The Amsterdam Compiler Kit (TACK; since 1980; `http://tack.sourceforge.net/`)
4. Java Virtual Machine (JVM; Sun; `http://en.wikipedia.org/wiki/Java_Virtual_Machine`)
5. Common Intermediate Language (CIL; Microsoft; `http://en.wikipedia.org/wiki/Common_Intermediate_Language`)

Language Structures I

Structures in imperative programming languages:
(object-oriented, declarative [functional/logic]: see special courses)

- Basic data types and basic operations
- Static and dynamic data structures
- Expressions and assignments
- Control structures (sequences, branching statements, loops, ...)
- Procedures and functions
- Modularity: blocks, modules, and classes

Use of procedures and blocks:

- FORTRAN: non-recursive and non-nested procedures
  $\Longrightarrow$ static memory management (memory requirement determined at compile time)
- C: recursive and non-nested procedures
  $\Longrightarrow$ dynamic memory management using runtime stack (memory requirement only known at runtime), no static links
- Algol-like languages (Pascal, Modula): recursive and nested procedures
  $\Longrightarrow$ dynamic memory management using runtime stack with static links

Language Structures II

Structures in machine code: (von Neumann/SISD)

Memory hierarchy: accumulators, registers, cache, main memory, background storage

Instruction types: arithmetic/Boolean/... operation, test/jump instruction, transfer instruction, I/O instruction, ...

Address modes: direct/indirect, absolute/relative, ...

Architectures: RISC (few [fast but simple] instructions, many registers), CISC (many [complex but slow] instructions, few registers)

Structures in intermediate code:

- Data types and operations like PL
- Data stack with basic operations
- Jumping instructions for control structures
- Runtime stack for blocks, procedures, and static data structures
- Heap for dynamic data structures

Outline

The Example Programming Language EPL

Structures of EPL:

- Only integer and Boolean values
- Arithmetic and Boolean expressions with strict and non-strict semantics
- Control structures: sequence, branching, iteration
- Nested blocks and recursive procedures with local and global variables
  ($\Longrightarrow$ dynamic memory management using runtime stack with static links)
- Procedure parameters and data structures later

Syntax of EPL

Definition 17.8 (Syntax of EPL)

The syntax of EPL is defined as follows:

$$
\begin{array}{lll}
\mathbb{Z}: & z & (*\ z \text{ is an integer } *) \\
Ide: & I & (*\ I \text{ is an identifier } *) \\
AExp: & A ::= z \mid I \mid A_1 \texttt{ + } A_2 \mid \ldots & \\
BExp: & B ::= A_1 \texttt{ < } A_2 \mid \texttt{not } B \mid B_1 \texttt{ and } B_2 \mid B_1 \texttt{ or } B_2 & \\
Cmd: & C ::= I \texttt{ := } A \mid C_1\texttt{;}C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \mid & \\
 & \quad \texttt{while } B \texttt{ do } C \mid I\texttt{()} & \\
Dcl: & D ::= D_C\ D_V\ D_P & \\
 & D_C ::= \varepsilon \mid \texttt{const } I_1 \texttt{ := } z_1\texttt{,}\ldots\texttt{,}I_n \texttt{ := } z_n\texttt{;} & \\
 & D_V ::= \varepsilon \mid \texttt{var } I_1\texttt{,}\ldots\texttt{,}I_n\texttt{;} & \\
 & D_P ::= \varepsilon \mid \texttt{proc } I_1\texttt{;}K_1\texttt{;}\ldots\texttt{;}I_n\texttt{;}K_n\texttt{;} & \\
Block: & K ::= D\ C & \\
Pgm: & P ::= \texttt{in/out } I_1\texttt{,}\ldots\texttt{,}I_n\texttt{;}K\texttt{.} &
\end{array}
$$