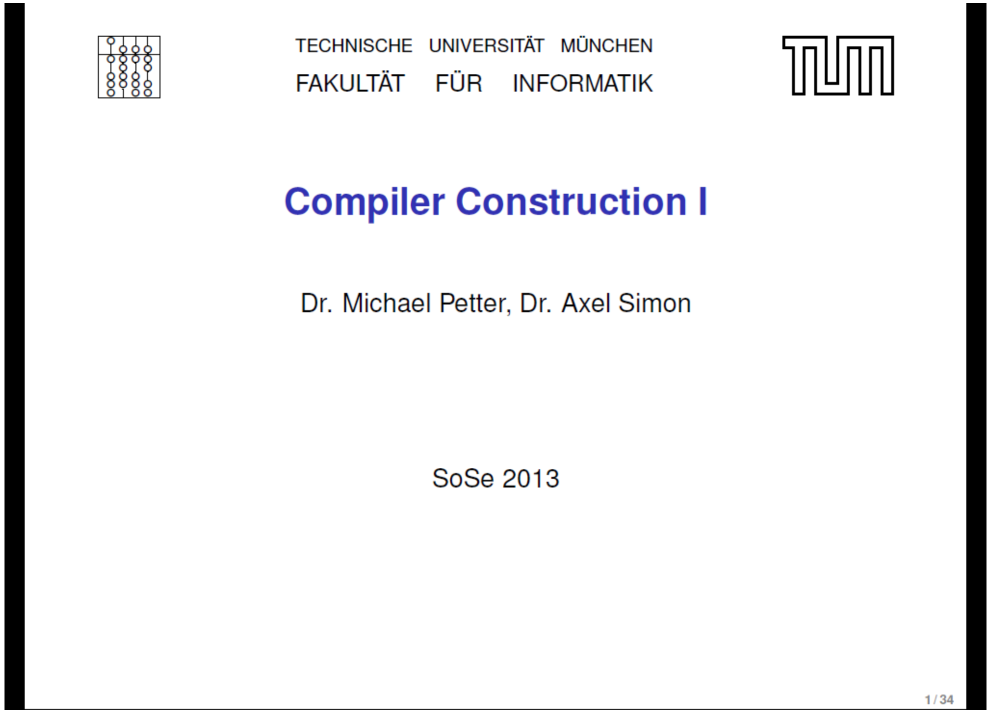
Script generated by TTT

Title: Simon: Compilerbau (03.06.2013)

Date: Mon Jun 03 14:52:29 CEST 2013

Duration: 52:56 min

Pages: 34

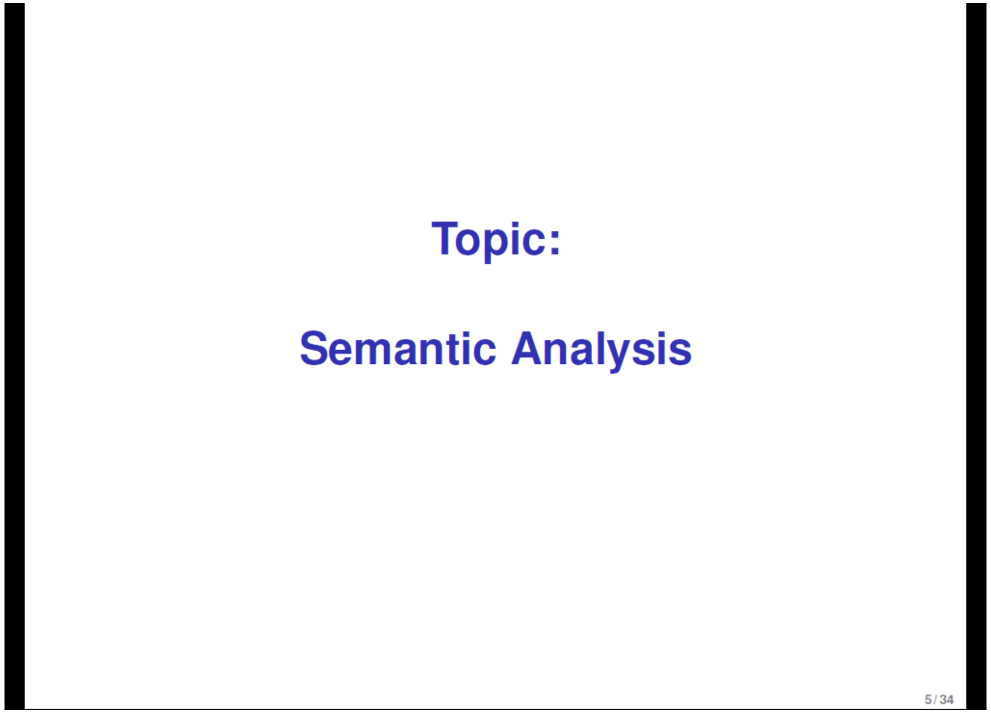


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Compiler Construction I

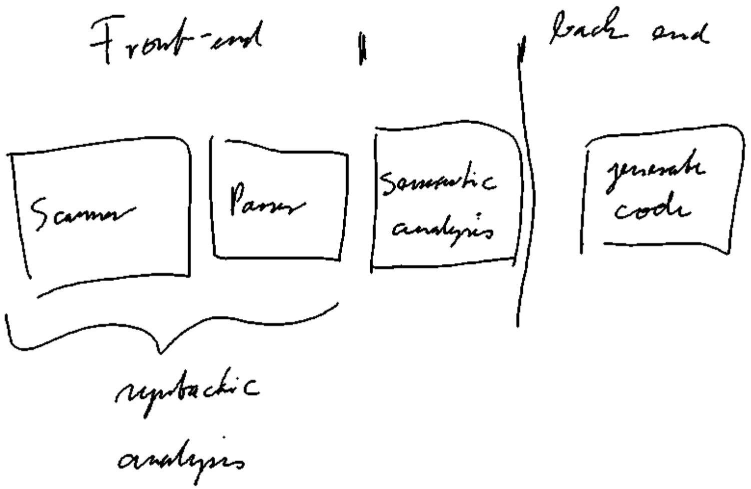
Dr. Michael Petter, Dr. Axel Simon

SoSe 2013



Topic:

Semantic Analysis



Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables
- semantic analyses are also useful to
  - find possibilities to “optimize” the program
  - warn about possibly incorrect programs

Semantic Analysis

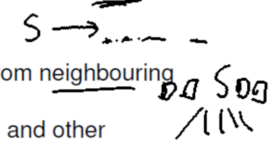
Scanner and parser accept programs with correct syntax.

- not all programs that are syntactically correct make *sense*
- the compiler may be able to *recognize* some of these
    - these programs are rejected and reported as erroneous
    - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
    - check that identifiers are known and where they are defined
    - check the type-correct use of variables
- semantic analyses are also useful to
    - find possibilities to “optimize” the program
    - warn about possibly incorrect programs

⇝ a semantic analysis annotates the syntax tree with attributes

Attribute Grammars

- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the *type* of that node (which is usually a non-terminal)
- we call this a *local* computation:
    - only accesses already computed information from neighbouring nodes
    - computes new information for the current node and other neighbouring nodes



Attribute Grammars

- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the *type* of that node (which is usually a non-terminal)
- we call this a *local* computation:
    - only accesses already computed information from neighbouring nodes
    - computes new information for the current node and other neighbouring nodes

Definition attribute grammar

An attribute grammar is a CFG extended by

- an set of attributes for each non-terminal and terminal
- local attribute equations

Attribute Grammars

- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the *type* of that node (which is usually a non-terminal)
- we call this a *local* computation:
    - only accesses already computed information from neighbouring nodes
    - computes new information for the current node and other neighbouring nodes

Definition attribute grammar

An attribute grammar is a CFG extended by

- an set of attributes for each non-terminal and terminal
- local attribute equations

- in order to be able to evaluate the attribute equations, all attributes mentioned in that equation have to be evaluated already

⇝ the nodes of the syntax tree need to be visited in a certain *sequence*

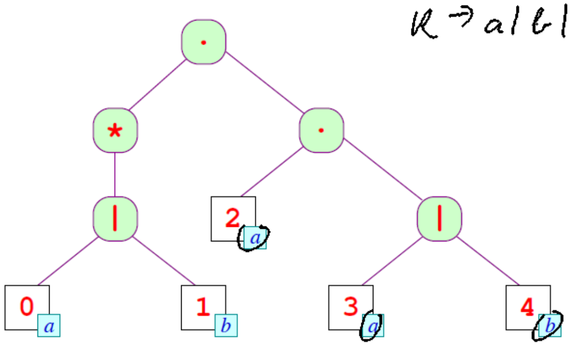
Example: Computation of the empty$[r]$ Property

Consider the syntax tree of the regular expression (a|b)*a(a|b):

S → R

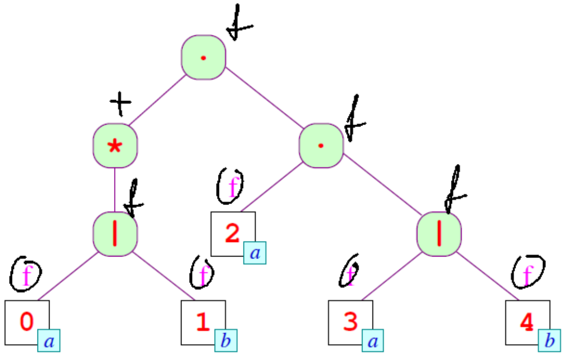
R → a | b | R|R | R·R

| R*



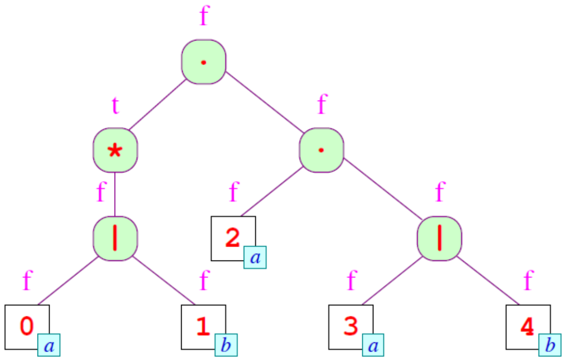
Example: Computation of the empty$[r]$ Property

Consider the syntax tree of the regular expression (a|b)*a(a|b):



Example: Computation of the empty$[r]$ Property

Consider the syntax tree of the regular expression (a|b)*a(a|b):



⇝ equations for empty$[r]$ are computed from bottom to top (aka bottom-up)

Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a *depth-first* traversal:
  - at a leaf, we can compute the value of empty without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the empty attribute is a *synthetic* attribute
- it may be computed by a ~~pre- or~~ post-order traversal

Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a *depth-first* traversal:
  - at a leaf, we can compute the value of empty without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the empty attribute is a *synthetic* attribute
- it may be computed by a pre- or post-order traversal

in general:

Definition

An attribute is called

- synthetic if its value is always propagated upwards in the tree (in the direction leaf → root)
- inherited if its value is always propagated downwards in the tree (in the direction root → leaf)

Attribute Equations for empty

In order to compute an attribute *locally*, we need to specify attribute equations for each node. rule in CFG
These equations depend on the *type* of the node:

rhs of rule

for leafs: $r \equiv \boxed{i \mid x}$ we define $\text{empty}[r] = (x \equiv \epsilon)$.

otherwise:

$$\begin{aligned}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \vee \text{empty}[r_2]\\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \wedge \text{empty}[r_2]\\
\text{empty}[r_1^*] &= t\\
\text{empty}[r_1?] &= t
\end{aligned}$$

Specification of General Attribute Systems

The empty attribute is *synthetic*, hence, the equations computing it can be given using *structural induction*.

Specification of General Attribute Systems

The empty attribute is *synthetic*, hence, the equations computing it can be given using *structural induction*.
In general, attribute equations combine information for children and parents.

- ⇝ need a more flexible way to specify attribute equations that allows mentioning of parents and children
- use consecutive indices to refer to neighbouring attributes

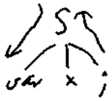
$\text{empty}[0]$:	the attribute of the current node
$\text{empty}[i]$:	the attribute of the i-th child $(i > 0)$

... in the example:

$$\begin{aligned}
\boxed{x} \quad &: \quad \text{empty}[0] := (x \equiv \epsilon)\\
\boxed{\mid} \quad &: \quad \text{empty}[0] := \text{empty}[1] \vee \text{empty}[2]\\
\boxed{\cdot} \quad &: \quad \text{empty}[0] := \text{empty}[1] \wedge \text{empty}[2]\\
\boxed{*} \quad &: \quad \text{empty}[0] := t\\
\boxed{?} \quad &: \quad \text{empty}[0] := t
\end{aligned}$$

Observations

S → var x i

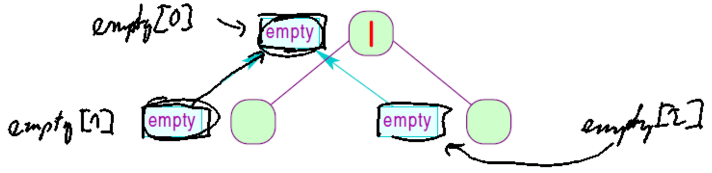


- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

Observations

- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

We illustrate dependencies between attributes using directed graph edges:



⇝ arrow points in the direction of information flow

Observations

S → T

S → R

- in order to infer an evaluation strategy, it is not enough to consider the *local* attribute dependencies at each node
- the evaluation strategy must also depend on the *global* dependencies, that is, on the the information flow between nodes
- the global dependencies thus change with each new abstract syntax tree
- in the example, the information flows always from the children to the parent node
  ⇝ a post-order depth-first traversal is possible
- in general, variable dependencies can be much more complicated

Simultaneous Computation of Multiple Attributes

Compute empty, first, next of regular expression:

$\boxed{x}$:

empty[0] := $(x \equiv \epsilon)$

first[0] := $\{x \mid x \neq \epsilon\}$

// (no equation for next)

root: :

empty[0] := empty[1]

first[0] := first[1]

next[0] := $\emptyset$

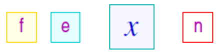
next[1] := ~~next[0]~~ ∅

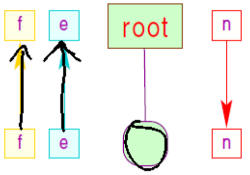
root → R

R → x | ε

| R · R

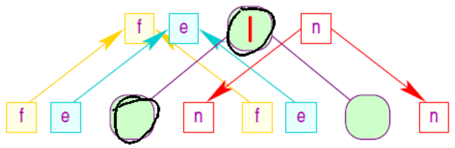
| R *





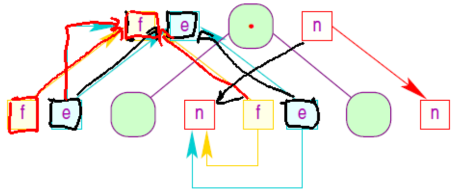
Regular Expressions: Rules for Alternative

| : empty[0] := empty[1] ∨ empty[2]
first[0] := first[1] ∪ (empty[1] ? first[2] : ∅)
next[1] := next[0]
next[2] := next [0]



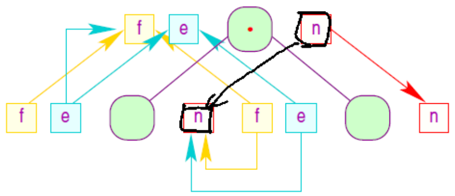
Regular Expressions: Rules for Concatenation

· : empty[0] := empty[1] ∧ empty[2]
first[0] := first[1] ∪ (empty[1] ? first[2] : ∅)
next[1] := (empty[2] ? next[0] : ∅) ∪ first[2])
next[2] := next[0]



Regular Expressions: Rules for Concatenation

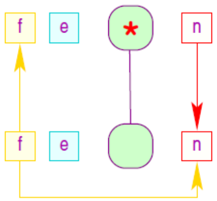
· : empty[0] := empty[1] ∧ empty[2]
first[0] := first[1] ∪ (empty[1] ? first[2] : ∅)
next[1] := first[2] ∪ (empty[2] ? next[0] : ∅)
next[2] := next[0]

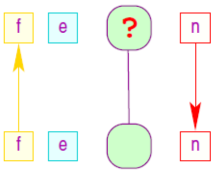


Regular Expressions: Kleene-Star and ‘?’

* : empty[0] := t
first[0] := first[1] ∪ next[0]
next[1] := next[0] ∪ first[1]

? : empty[0] := t
first[0] := first[1] ∪ next[0]
next[1] := next[0] ∪ first[1]

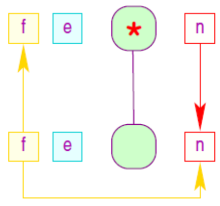


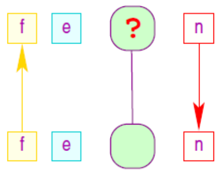


Regular Expressions: Kleene-Star and '?'

$$\boxed{*} \; : \quad \begin{array}{lcl} \text{empty}[0] & := & t \\ \text{first}[0] & := & \text{first}[1] \cup \text{next}[0] \\ \text{next}[1] & := & \text{first}[1] \cup \text{next}[0] \end{array}$$

$$\boxed{?} \; : \quad \begin{array}{lcl} \text{empty}[0] & := & t \\ \text{first}[0] & := & \text{first}[1] \cup \text{next}[0] \\ \text{next}[1] & := & \text{next}[0] \end{array}$$





Challenges for General Attribute Systems

- an evaluation strategy can only exist if for *any* abstract syntax tree, the dependencies between attributes are acyclic
- checking that no cyclic attribute dependencies can arise is DEXPTIME-complete [Jazayeri, Odgen, Rounds, 1975]

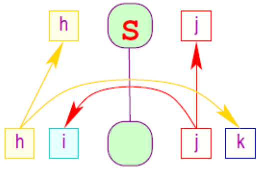
Idea: Compute a *set* of dependency graphs for each symbol $s \in T \cup N$.

- Initialize $G(s) = \emptyset$ for each $s \in N$ and set $S(s) = \{G_s\}$ for each $s \in T$ where G_s is the dependency graph of s.
- For each rule $s ::= s_1 \ldots s_n$ of the non-terminal $s \in N$ mit RHS $s_1 \ldots s_n$ extend $G(s)$ with graphs obtained by embedding the dependency graphs $G(s_1), \ldots G(s_n)$ into the child attributes of the dependency graph of that rule.

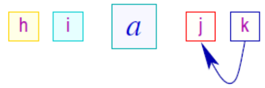
Computing Dependencies

Example: Given the grammar $S ::= a \mid b$ with these dependencies:

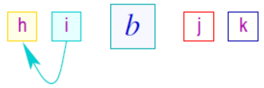
G_s



G_a



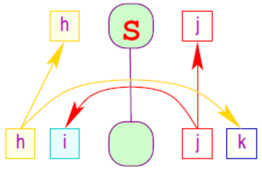
G_b

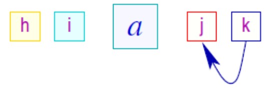


Start with $G(S) = \emptyset$, $G(a) = \{k[0] \to j[0]\}$, and $G(b) = \{i[0] \to h[0]\}$.

Computing Dependencies

Example: Given the grammar $S ::= a \mid b$ with these dependencies:

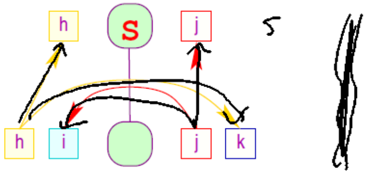


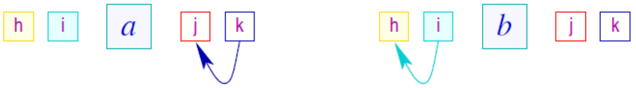


Start with $G(S) = \emptyset$, $G(a) = \{k[0] \to j[0]\}$, and $G(b) = \{i[0] \to h[0]\}$.

Computing Dependencies

Example: Given the grammar $S ::= a \mid b$ with these dependencies:





Start with $G(S) = \emptyset$, $G(a) = \{k[0] \to j[0]\}$, and $G(b) = \{i[0] \to h[0]\}$. For rule $\underline{S ::= a}$, embed $G(a)$ into the child attributes of rule $S ::= a$, yielding

$$G'(S) = \{\underline{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0]}, \underline{k[I] \to j[I]}\}$$

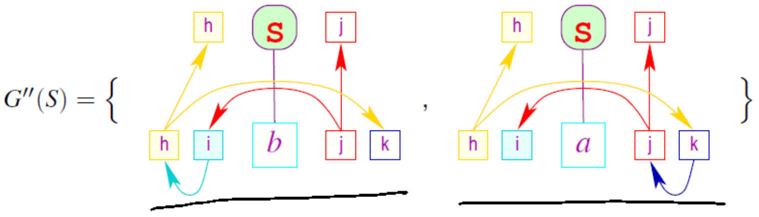
Computing Dependencies (cont'd)

Result so far:

$$G'(S) = \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], k[I] \to j[I]\}$$

Given rule $S ::= b$, embed $G(b)$ into the child attributes of rule $S ::= a$, yielding

$$G''(S) = G'(S) \cup \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], i[I] \to h[I]\}$$



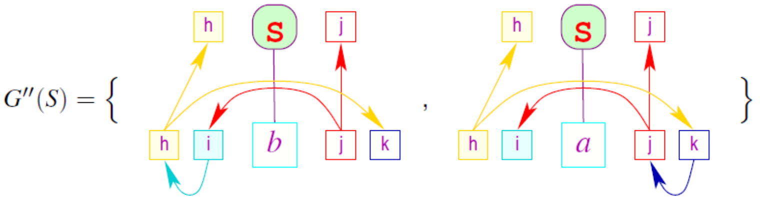
Computing Dependencies (cont'd)

Result so far:

$$G'(S) = \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], k[I] \to j[I]\}$$

Given rule $S ::= b$, embed $G(b)$ into the child attributes of rule $S ::= a$, yielding

$$G''(S) = G'(S) \cup \{h[1] \to h[0], h[1] \to k[1], j[1] \to i[1], j[1] \to j[0], i[I] \to h[I]\}$$



None of the graphs in G'' contain a cycle $\leadsto$ every derivable abstract syntax tree can be evaluated.