A SYNTAX-ANALYZER CONSTRUCTOR

EUGENE J. ROLLINS

Oregon Graduate Center

OGC Technical Report No. CS/E-82-05

A Syntax-Analyzer Constructor

Eugene J. Rollins
Oregon Graduate Center
Beaverton, Oregon

(Extended Abstract - 12 August 1982)

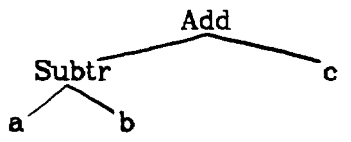
1. Introduction

Most programming languages (pure LISP excepted) are given a syntax intended to make programs easy for the human reader to absorb, at least superficially. These syntactic constructs alleviate the need to count parentheses, by adopting various conventions such as infix operator notation and operator precedence rules, in order to denote the association of operand expressions with their operator symbols. In the internal representation of programs for machine translation or interpretation, one invariably prefers a simpler, more uniform convention to bind operands to operators. Such a convention is embedded in a very uniform notation called ***abstract syntax trees*** (ASTs), commonly used in a great many compilers, structure editors and related language-knowledgeable software tools. Since the syntax of both a concrete programming language and ASTs can be formally defined by deterministic, context-free grammars, it seems reasonable to expect that a translation from one to the other could be derived automatically.

I have extended the concept of a parser constructor to that of a ***syntax-analyzer constructor***. A ***syntax analyzer***, which includes lexical analysis, parsing, and association of operands to operators, translates source language programs to ASTs. The syntax-analyzer constructor accepts a simple description of the translation from source programs to ASTs. From this input it automatically produces a syntax analyzer that implements this translation as well as providing feed-back to the user. This feedback, given in the form of a grammar, describes the class of ASTs produced by the constructed syntax-analyzer. Before we can discuss this constructor we need to define a few terms related to abstract syntax.

The root of an AST is labeled by an operator whose operands are represented by the subtrees. Leaves are operand terminal symbols such as identifiers or numbers. An AST may be displayed linearly or graphically as shown below.

<Add, <Subtr, <a, b>>, c>



An ***abstract grammar*** describes a set of ASTs. Consider for example, the abstract grammar given in Table 1. In this grammar Type and Expr are ***syntactic domains***. A syntactic domain, S, denotes a set of ASTs, A_S. If $x \in S$ then all ASTs having x-labeled roots are members of A_S. In the example, A_{Expr} includes any AST whose root is one of Identifier, Add or Subtr. The ***productions*** describe the form of ASTs. An AST with a root labeled Add has two subtrees both of which are members of A_{Expr}. Note that there is just one production for any operator (nonterminal) of an abstract grammar.

Formally, every abstract grammar, A, gives rise to an ***initial S-sorted Σ-Algebra***, I. Each syntactic domain and nonterminal of A corresponds to a sort of I. The terms of the algebra are ASTs defined by A. An in-depth discussion of abstract syntax is given in [Rol82a], where this correspondence between abstract grammars and initial-algebras is defined. See [Gog77a], for more information on initial-algebra semantics.

Syntactic Domains
Type = {Product, Identifier}
Expr = {Identifier, Add, Subtr}

Productions

Constant	→	Identifier Type Expr
Product	→	Type Type
Add	→	Expr Expr
Subtr	→	Expr Expr

Table 1

A syntax-analyzer constructor accepts as input a CFG and a specification of a translation from parse trees to ASTs. Its output, of course, is a syntax-analyzer that translates a source language program into an AST. Input to this syntax-analyzer constructor, *Sac*, is a CFG whose productions and terminal symbols have been annotated to specify the translation from parse trees to ASTs.

Each semantically significant terminal symbol is annotated. Every occurrence of an annotated terminal in a parse tree is to be retained as a leaf node in the target AST. Significant terminals typically include identifiers and numeric constants, but not keywords, parentheses, commas or other punctuation.

A parse tree whose root is expanded by an annotated production is translated to an AST of the class specified by the annotation.

For any annotated grammar, G, there is a *corresponding abstract grammar*, $\hat{G}$, which describes the class of ASTs produced by the syntax-analyzer constructed from G. The full-length paper will contain an algorithm for computing $\hat{G}$ given G. This abstract describes an algorithm for producing a syntax-analyzer from an annotated parsing grammar.

By examining $\hat{G}$ one can check to see if an annotated grammar denotes the intended translation from source programs to ASTs. Any software tool that uses ASTs as an internal representation of programs can be driven by tables describing the abstract grammar. A large part of a programming language environment can be synthesized from skeleton software tools.

The semantics of annotated grammar is expressed in a variant of Backus's functional language FP [Bac78a]. What makes FP an interesting language is that one can do programming and mathematical reasoning in the same language. An FP system comes with an algebra of programs that is no more difficult to use than high-school algebra. Since the meanings of annotations are defined via FP, the AST-building function denoted by G is easily cast in FP. Through algebraic manipulation of the resulting FP function, $\hat{G}$ can also be derived.

A brief review of FP systems and the definition of the particular FP system used in this paper appears in the appendix.

2. The relation between concrete and abstract syntax

The grammar annotations allow one to identify operators of an abstract grammar with certain productions of a CFG that define the concrete syntax of a language. Since operands may not be associated with operators in a uniform manner by the concrete syntax, various syntactic operand-association rules must be applied to construct operand lists for operators. The annotations allow one to associate with each terminal symbol and each production of the CFG a function from lists of (operand) ASTs to lists of (operand) ASTs.

There are two kinds of annotations: *terminal annotations* and *production annotations*. A terminal annotation declares a set of terminal symbols as semantically significant operands. Identifiers and numeric constants are typical examples of such terminals. A terminal annotation has the form

$terminal $t_1, t_2, \ldots, t_m$
(where t_i is a terminal symbol of the CFG)

A production annotation may be attached to a production P to identify particular operator of the abstract grammar with each node of a parse tree expanded by P. This annotation has the form

$operator X

(where X is the name of an operator of the abstract grammar)

The syntax for annotated grammar is given in Table 2. The meanings of these annotations are informally explained through an example. A formal definition soon follows. Figure 3 gives the annotated grammar for a small example language, and a sample source program in that language with a parse tree and an AST.

Syntax of the Sac Input Language

SyntaxDescription	$\rightarrow$ {TerminalAnnotation} AnnotatedProduction$^+$
TerminalAnnotation	$\rightarrow$ $terminal Identifier$^+$
AnnotatedProduction	$\rightarrow$ CFGProduction {Annotation}
Annotation	$\rightarrow$ $operator Identifier {PrefixOperands}
PrefixOperands	$\rightarrow$ (Number)

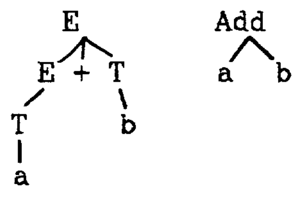
Note: CFGProduction is a context-free grammar production and has the usual form [Hop79a].

Table 2

	$terminal Identifier	
1	E $\rightarrow$ E + T	$operator Add
2	E $\rightarrow$ E − T	$operator Subtr
3	E $\rightarrow$ T	
4	T $\rightarrow$ Identifier	
5	T $\rightarrow$ (E)	

Annotated Grammar

a + b



Source Program **Parse Tree** **AST**

Figure 3

The first annotation tells us that Identifier is the only semantically significant terminal token; all others are either operator symbols or punctuation marks. The annotations attached to productions 1 and 2 identify those productions as corresponding to the occurrence of semantic operators. Names (Add and Subtr) are given to these operators. Let us consider how an AST is built from the parse tree of Figure 3.

ASTs can be built bottom-up. In a post-order traversal of this parse tree, an identifier, a, is first encountered. Since identifiers are semantically significant, "a" is saved in a list of operands. Productions 4 and 3 are found to expand the interior nodes visited next. But, since these are unannotated the list of operands <a> is unmodified. Since + is not mentioned in the terminal annotation it is ignored. Translating the root's last subtree yields the list <b>. The operand lists derived from the subtrees are concatenated, forming <a, b>. Production 1, which expands the root, has associated with it the operator symbol Add. An AST is built from the operand list which has been gathered, using the operator Add to label its root, to give the AST shown in Figure 3.

3. The semantics of annotated grammars

Annotated grammars have a formal semantics. Let ASTlist be the set of all sequences of ASTs. For each annotated grammar, G, a function τ :ParseTrees $\rightarrow$ ASTlist $\rightarrow$ ASTlist is defined. A parse tree is either a leaf representing a terminal symbol of G or a node expanded by a production of G. τ is defined structurally over parse trees by associating a function from ASTlist → ASTlist with every terminal symbol and production of G. These functions are presented below. Before proceeding with their definition a comment on notation is in order.

Σ_G is a set of functions indexed by productions of G, each of which maps a tuple of parse trees into a new parse tree. These are defined such that

$$\sigma_{A \to \alpha_1 \cdots \alpha_n}(\vartheta_1, \ldots, \vartheta_n)$$

denotes the parse tree whose root is expanded by the production $A \to \alpha_1 \cdots \alpha_n$ and whose subtrees are $\vartheta_1, \ldots, \vartheta_n$.

In terms of FP objects, an AST is represented by a sequence

$$\blacktriangleleft x, \langle t_1, \ldots, t_n \rangle \blacktriangleright$$

where x is the label of the root and $t_1, \ldots, t_n$ are objects representing the subtrees. Typed brackets ($\blacktriangleleft\blacktriangleright$) are used to distinguish ASTs from untyped sequences. To distinguish functions that construct ASTs from untyped construction we shall replace the usual [ ] by ⟦ ⟧. All functions and combining forms are polymorphic; they handle typed and untyped construction uniformly.[†]

Terminal Symbols

Let t be a terminal symbol of the CFG.

(1) **annotation:** $terminal t
define $\tau(t) \equiv \text{apndr} \circ [\text{id}, \bar{t}]$
Any terminal listed as being a semantically significant operand is added to the list of operands being constructed.

(2) no annotation for t
define $\tau(t) \equiv \text{id}$
Any unannotated terminal is ignored.

(3) The symbol λ indicates the (null) right-hand side of an empty production.
define $\tau(\lambda) \equiv \text{id}$
The operand list is left unchanged.

Productions

Let $A \to \alpha_1 \cdots \alpha_n$ be a production of the CFG.

(4) unannotated production
define $\tau(\sigma_{A \to \alpha_1 \cdots \alpha_n}(\vartheta_1, \ldots, \vartheta_n)) \equiv \tau(\vartheta_n) \circ \cdots \circ \tau(\vartheta_1)$
The subtrees are traversed from left to right. The operand list may be augmented by the concatenation of additional operands, if any are produced by the subtrees. The resulting list is simply returned.

(5) **annotation:** $operator X
define $\tau(\sigma_{A \to \alpha_1 \cdots \alpha_n}(\vartheta_1, \ldots, \vartheta_n)) \equiv \text{apndr} \circ [\text{id}, [\![\bar{X}, \tau(\vartheta_n) \circ \cdots \circ \tau(\vartheta_1) \circ \bar{\varphi}]\!]]$
An AST is created from the operator X and the operand list constructed by traversing the subtrees of P from left to right. This AST is added to the operand list.

Given τ, syntax analysis proceeds as follows. The source program, P, is parsed giving the parse tree, T_P. $\tau(T_P)$ yields ψ_P :ASTlist → ASTlist. The application $\psi_P \bullet \varphi$ gives the AST for P.

Example

Below the function ψ_{a+b} :ASTlist → ASTlist is given for the parse tree of Figure 3 and is simplified.

† In defining the semantics of annotated grammars, only a few FP functions are used. For the reader not intimately familiar with FP, it may be helpful to think of an abstract data type *stack*, and to read apndr as *push*, 1r as *top*, and tlr as *pop*.

$$\psi_{a+b} \equiv \tau(\sigma_{E \to E+T}(\sigma_{E \to T}(\sigma_{T \to Id}(\sigma_a)), \sigma_+, \sigma_{T \to Id}(\sigma_b)))$$
$$\equiv \text{(by expanding definitions)}$$
$$\text{apndr} \circ [\text{id}, [\![\overline{\text{Add}}, \text{apndr} \circ [\text{id}, \bar{b}] \circ \text{apndr} \circ [\text{id}, \bar{a}] \circ \bar{\varphi}]\!]]$$
$$\equiv \text{(by two applications of Law 10)}$$
$$\text{apndr} \circ [\text{id}, [\![\overline{\text{Add}}, [\bar{a}, \bar{b}]]\!]]$$

Applying $\psi_{a+b} \bullet \varphi$ gives the sequence that contains a single element: the AST in Figure 3.

$$\psi_{a+b} \bullet \varphi \equiv \langle\blacktriangleleft\text{Add}, \langle a, b\rangle\blacktriangleright\rangle$$

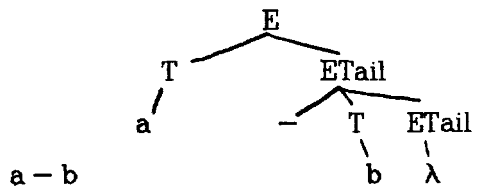
This completes the example showing a parse tree to AST translation.

3.1. Extended annotations accommodate LL(1) parsing

Note that in definition (5) above the operand list constructed for operator X starts empty. This means that an AST is an operand of X if and only if it is constructed from the subtrees of P. This restriction would prohibit the use of LL(1) parsing. Figure 4 contains a naively annotated LL(1) grammar that recognizes the same language as the grammar shown in Figure 3.

	$terminal Identifier		
1	E	→ T ETail	
2	ETail	→ − T ETail	$operator Subtr
3	ETail	→ + T ETail	$operator Add
4	ETail	→ λ	
5	T	→ Identifier	
6	T	→ (E)	

Annotated Grammar



Input **Parse Tree**

Figure 4

Using the annotated grammar of Figure 4 we could attempt to produce an AST from "a−b". In doing this we get unexpected results.

$$\psi_{a-b} \bullet \varphi \equiv \langle a, \blacktriangleleft\text{Subtr}, \langle b\rangle\blacktriangleright\rangle$$

This list of two ASTs is not the desired result. We want

$$\langle\blacktriangleleft\text{Subtr}, \langle a, b\rangle\blacktriangleright\rangle$$

Due to the 1-symbol lookahead of LL(1) parsing, infix operator symbols must be head-symbols of productions. This causes the problem we encountered above. Let us examine the translation of the parse tree of Figure 4. The first subtree produces the AST comprising the leaf "a". The AST $\blacktriangleleft$Subtr, <b>$\blacktriangleright$ is formed from translation of the second subtree. Since the production expanding the root is unannotated these ASTs are combined into a list of two ASTs. But, the leaf "a" should be attached as a subtree of the second AST.

To solve this problem, the production annotation has been extended to handle prefix operands of infix or postfix operators. The annotation

$operator X (n)

indicates that there are n prefix operands. The formal definition of this annotation is given below.

Extended Annotation of Productions

Let $A \to \alpha_1 \cdots \alpha_n$ be a production of the CFG.

(6) **annotation:** \$operator X (1)
define $\tau(\sigma_{A \rightarrow \alpha_1 \cdots \alpha_n}(\vartheta_1, \ldots, \vartheta_n)) \equiv \mathrm{apndr} \circ [\mathrm{tlr}, \llbracket\bar{X}, \tau(\vartheta_n) \circ \cdots \circ \tau(\vartheta_1) \circ [1r]\rrbracket]$
In both (5) and (6), an AST with X at its root is appended to the operand list being constructed. A new operand list is built for X. In (5), this list starts empty. But in (6), the rightmost operand of the list being constructed is removed and is used to head the new operand list for X.

(7) **annotation:** \$operator X (m)
define $\tau(\sigma_{A \rightarrow \alpha_1 \cdots \alpha_n}(\vartheta_1, \ldots, \vartheta_n))$
$$\equiv \mathrm{apndr} \circ [\mathrm{tlr}^m, \llbracket\bar{X}, \tau(\vartheta_n) \circ \cdots \circ \tau(\vartheta_1) \circ [mr, m{-}1r, \ldots, 1r]\rrbracket]$$

With these new annotations, the grammar of Figure 4 can be corrected by appending "(1)" to each production annotation. Now consider the translation of the parse tree of Figure 4 to an AST. The second subtree of this parse tree is expanded by production 2. The simplified translation function corresponding to this subtree is

$$\mathrm{apndr} \circ [\mathrm{tlr}, \llbracket\overline{\mathrm{Subtr}}, \mathrm{apndr} \circ [\mathrm{id}, \bar{b}] \circ [1r]\rrbracket]$$

For the first subtree, we get

$$\mathrm{apndr} \circ [\mathrm{id}, \bar{a}]$$

The production that expand the root is unannotated, therefore we compose the above functions, simplify and get

$$\mathrm{apndr} \circ [\mathrm{id}, \llbracket\overline{\mathrm{Subtr}}, \mathrm{apndr} \circ [\mathrm{id}, \bar{b}] \circ [\bar{a}]\rrbracket]$$

The above function illustrates that the prefix operand is integrated into the construction of the Subtr AST. Applying the above function to φ yields $\blacktriangleleft$Subtr, <a, b>$\blacktriangleright$.

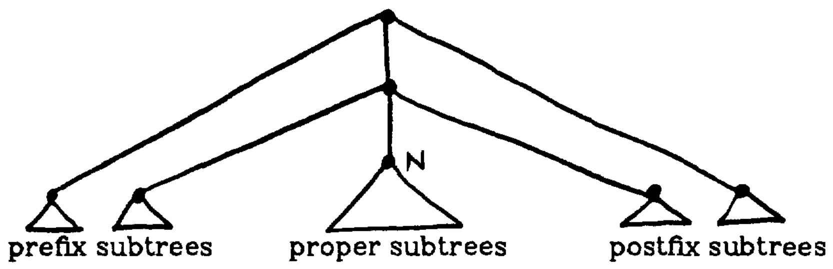
4. Annotated grammars are adequate for description of syntax analyzers

Annotated grammars as defined by (1) - (7) is complete enough to describe translation from source language programs to ASTs if we make the following assumptions:

1- Syntactically, the source language is a deterministic context-free language.

2- Semantically, the source language adheres to the *principle of compositionality*. This principle states that the meaning of a phrase is a function of the meanings of its constituent phrases.

Consider the parse tree below produced by a CFG, G. Let P be an annotated production of G that expands node, N, which is labeled in the diagram. We can partition subtrees of this parse tree into three classes with respect to N; prefix, proper and postfix subtrees of N are labeled in the diagram. Let X be the operator associated with P.



The above assumptions guarantee that we can write G in such a way that the operands for X will be produced in the prefix and proper subtrees of N. The annotations define functions that gather operands from these subtrees when applied to parse trees

5. Experience

Sac, the syntax-analyzer constructor defined here, has been implemented and has proven to be a valuable software tool. We have been using Sac to build translators for experimental programming languages for over a year. These languages tend to be moving targets; their definitions are usually adjusted before an implementation is completed. After making the required changes to the annotated grammar, a new syntax analyzer is reconstructed whenever need. Modifiablity is critical when developing prototype software. By using ASTs as an intermediate language, we have separated syntax analysis and semantic interpretation in such a way that minor changes in one phase do not force changes in the other.

Sac has been used successfully by students in a compiler design course. This allowed more time for studying previously underemphasized aspects of compiler design such as type-checking, optimization and code generation.

A production-quality implementation of this constructor has been developed in industry based on our prototype. It has been used in the development of production-quality compilers and as the basis for a syntax-directed editor constructor ,[Bar82a].

Sac produces a syntax analyzer that includes LL(1) parsing. Developing an annotated LL(1) grammar can at first be a bit tricky. This task becomes much easier with some experience. It would be helpful if Sac would accept an arbitrary context-free grammar and produce an LL(1) grammar for us. Unfortunately, it is undecidable whether or not an arbitrary context-free grammar generates an LL(k) language, even for a fixed k [Ros70a].

References

Bac78a. J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.," *CACM* **21**(8)(August 1978).

Bar82a. W. Barabash and D. G. Frank, "Automatic Generation of a Language Oriented Editor From an Annotated Context-Free Grammar," Digital Equipment Corp., Nashua, NH (Jan. 1982).

Gog77a. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial Algebra Semantics and Continuous Algebras," *JACM* **24**(1) pp. 68-95 (Jan. 1977).

Hop79a. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass. (1979).

Rol82a. E. J. Rollins, "Abstract Syntax in Theory and Practice," CS/E-82-04, Oregon Graduate Center, Beaverton, Or (July 1982).

Ros70a. D. J. Rosenkrantz and R. E. Stearns, "Properties of Determinitic Top-down Grammars," *Info. and Control* **17** pp. 226-256 (1970).

Appendix

1. A review of FP

An FP System <O, P, Γ> comprises the following:

- a set O of ***objects***
- a set P of ***primitive functions*** that map objects to objects
- a set Γ of ***functional forms*** that are used to combine existing functions or objects to form new functions
- Γ may also include the functional form ***constant***, usually written as a bar over a single argument. If $x \in O$, $\bar{x}$ is a function that returns x when applied to any argument.

For an FP system <O, P, Γ>, a set of functions F is defined by the following:

- $F \supseteq P$
- if $\gamma \in \Gamma$, $a_1, \ldots, a_n \in F$ and γ takes n arguments, then $\gamma(a_1, \ldots, a_n) \in F$
- if ***constant*** $\in \Gamma$ and $x \in O$, then $\bar{x} \in F$
- if the function f is defined by

    def f ≡ r

    (where $r \in F$ or r is a functional form applied to arguments)

  then $f \in F$.

The application of $f \in F$ to $x \in O$ is written $f \bullet x$.

2. A small FP system

When using FP for reasoning, it is more convenient to use laws as equivalences. Since it is not used here for computation, evaluation rules are immaterial. Therefore, the FP system defined below has non-strict semantics.

2.1. Objects

Objects are defined with BNF using the meta-symbols → | + * { }. The meta-symbols + and * are Kleene plus and times respectively. Any expression within { } is optional.

object → atom | sequence | ⊥
atom → digit$^+$ | letter {letter | digit}*
sequence → < {objectlist} >
objectlist → object {, object}*

The symbol ⊥ , called "bottom", means undefined. The empty sequence, < >, is also written φ. Examples of objects include

$\langle\langle A\rangle\rangle$ $\langle\varphi, A, \langle 6, \varphi\rangle\rangle$ $\perp$ AB $\langle\perp, \text{Test1}\rangle$ φ

2.2. Primitive functions

right selectors

1r : sequence → sequence
    1r ▪ x ≡ if x = $\langle x_1, \dots, x_n\rangle$ then x_n else ⊥ end
and for any positive integer m
mr : sequence → sequence
    mr ▪ x ≡ if x = $\langle x_1, \dots, x_n\rangle$ and n≥m then x_{n-m+1} else ⊥ end

other sequence functions

```
tlr : sequence → sequence
    tlr ▪ x ≡ if x = $\langle x_1\rangle$ then $\varphi$
        elseif x = $\langle x_1, \dots, x_n\rangle$ and n≥2 then $\langle x_1, \dots, x_{n-1}\rangle$
        else ⊥ end
apndr : sequence × O → sequence
    apndr ▪ x ≡ if x = $\langle\varphi, z\rangle$ then $\langle z\rangle$
        elseif x = $\langle\langle x_1, \dots, x_n\rangle, z\rangle$ then $\langle x_1, \dots, x_n, z\rangle$
        else ⊥ end
```

identity

id : O → O
    id ▪ x ≡ x

2.3. Functional forms

composition

∘ : F × F → F
    (f ∘ g) ▪ x ≡ f ▪ (g ▪ x)
    I may abbreviate f ∘ f ∘ · · · ∘ f as f^n where n is the number of times f appears.

construction

[] : F^* → F
    $[f_1, \dots, f_n]$ ▪ x ≡ $\langle f_1$ ▪ x, . . . , f_n ▪ x$\rangle$

constant

$\bar{}$: O → F
    $\bar{x}$ ▪ y ≡ x

2.4. Some useful laws of the FP algebra

Law 1. $f \circ (g \circ h) \equiv (f \circ g) \circ h$
Law 2. $\bar{\varphi} \equiv []$
Law 3. $\bar{x} \circ f \equiv \bar{x}$
Law 4. $[f_1, \dots, f_n] \circ g \equiv [f_1 \circ g, \dots, f_n \circ g]$
Law 5. $f \circ id \equiv id \circ f \equiv f$
Law 6. $tlr \circ apndr \circ [f, g] \equiv f$
Law 7. $1r \circ apndr \circ [f, g] \equiv g$
Law 8. $1r \circ [f] \equiv f$
Law 9. $apndr \circ [[f_1, \dots, f_n], g] \equiv [f_1, \dots, f_n, g]$
Law 10. $apndr \circ [id, \bar{x}] \circ [f_1, \dots, f_n] \equiv [f_1, \dots, f_n, \bar{x}]$