January 1982

Abstract syntax in theory and practice

Eugene J. Rollins

ABSTRACT SYNTAX IN THEORY AND PRACTICE

EUGENE J. ROLLINS

Oregon Graduate Center
OGC Technical Report No. CS/E-82-04

Abstract Syntax in Theory and Practice

Eugene J. Rollins
Oregon Graduate Center

ABSTRACT

The use of abstract syntax in the theory and practice of language translation is explored. A formal definition of abstract grammar is provided that shows how every abstract grammar gives rise to an initial algebra. The polymorphic operators typically found in practical abstract grammars are handled in a straightforward manner in this semantic definition.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *semantics; syntax*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *Algebraic approaches to semantics*; D.3.4 [**Programming Languages**]: Processors – *Compilers*

General Terms: Languages, Formal Definitions, Compilers

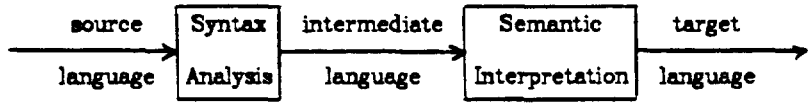
Additional Key Words and Phrases: Abstract syntax trees, initial algebras, polymorphism

1. Syntax and Semantics

The relationship between syntax and semantics has never been precisely defined. Some draw the distinction between syntax and semantics arbitrarily based on "what can be done at compile time" [Pag81a, pg. 76]. Those working on mathematically based semantic definition have generally ignored syntax and its relationship to semantics. They simply state that they give semantics to abstract syntax, reference McCarthy [McC63a] and leave it at that. The need for a well-defined relationship between syntax and semantics increases as theory filters into practice. Here this relationship is explored.

Author's address: Oregon Graduate Center, 19600 NW Walker Road, Beaverton, OR 97006

A good decomposition of a problem includes a precise definition of its subproblems and a description of how their solutions are composed to solve the original problem. I have decomposed language translation into two phases: syntax analysis and semantic interpretation, as shown in Figure 1.1. This section discusses the decomposition of language translation informally and pragmatically and shows how abstract syntax is used as an interface between these phases. The next section relates those ideas to approaches to formal semantic definition and gives a formal definition of abstract syntax trees through initial-algebra semantics. This formulation extends and refines sections 3.1 and 3.2 of [Gog77a]. In particular, this formulation handles the polymorphic operators typically found in practical abstract syntax.



Although this paper provides formal definitions, ideas will be presented informally through a common example. The example consists of a programming language fragment that comprises constant declarations, type expressions and (value) expressions. Figure 1.1 contains a context-free grammar (CFG) that defines the syntax of this fragment. Also given are two sample programs recognized by the CFG.

Grammar

ConstDecl	→ const Identifier : Type == Expr
Type	→ CartesianProduct | SimpleType
CartesianProduct	→ Type * SimpleType
SimpleType	→ Identifier | (Type)
Expr	→ Expr + Term | − Term | Term
Term	→ Term * Factor | Factor
Factor	→ Identifier | Numeral | <Expr, Expr> | (Expr)

Sample Programs

const A : integer == C * 4 + 3

const B : integer * boolean == <4, true>

Figure 1.1

A source language features one of many different concrete notations that may be chosen to manifest the same semantics. For example, as McCarthy [McC63a] points out, integer addition can be cast into several notations: "a+b", "+ab", "(PLUS A B)", "7^a11^b", and

others. An intermediate language notation should abstract only semantically significant details from programs.

The ***abstract syntax*** [McC63a] of a program specifies the operators that are used to express the semantics as well as the operands to which these operators are applied. It provides all the information that is required for semantic interpretation of that program. Abstract syntax may be represented through ***abstract syntax trees*** (ASTs).

Representation of programs as ASTs has been found convenient for use as an intermediate representation in translators because it abstracts only semantically significant details of concrete notation. Informally, the root of an AST represents an operator whose operands are represented as the subtrees of that AST. ASTs for the sample programs of Figures 1.1 are provided in Figure 1.2.

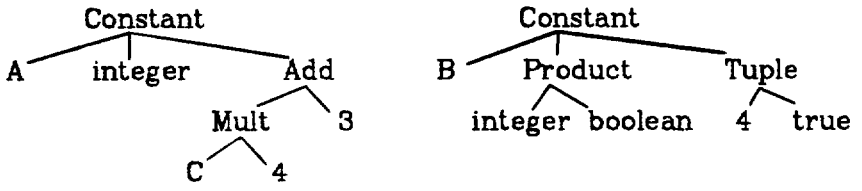


Figure 1.2

Let us examine how ASTs differ from concrete notations such as the one given in Figure 1.1. To compare them, let us first linearize the ASTs. A linear form can be given for any AST by listing a pre-order traversal of the AST. Figure 1.3 shows linear forms for the ASTs of Figure 1.2.

```
Constant (A, integer, Add (Mult (C, 4), 3))
Constant (B, Product (integer, boolean), Tuple (4, true))
```

Figure 1.3

The linear abstract syntax is a fully parenthesized notation and has uniform operand association rules. That is, the operator always appears in the prefix position, and its operands are elements of a parenthesized list to the right of the operator.

Association rules and redundant symbols are introduced into a concrete notation to make programs easier to read. Programs need not be fully-parenthesized; operator precedence rules disambiguate operand association. The choice of a concrete notation to express a particular meaning is guided by its familiarity to human readers. It need not have uniform association rules; infix, postfix, and prefix operator notations are commonly mixed in the same concrete notation. Different operators within the same concrete notation may associate to the left or right.

An intermediate language intended for semantic interpretation should be uniform in its rules for associating operands to operators. The description of operand association rules has no proper place in the semantic description of a language.

One might argue for the use of parse trees of the source language grammar as an intermediate language. However, parse trees are not only dependent upon the concrete notation selected, but on the chosen parsing method as well. There are several CFGs that recognize a context-free language, each with its own set of parse trees. Different parsing methods may require different grammars. Should the presentation of the language semantics demand alteration if the parsing method changes? I think not. A standard semantic definition that assumes a particular parsing method is of little use to a compiler designer interested in using a different parsing method.

ASTs are a natural starting point for semantic interpretation because they are independent of syntactic considerations such as parsing method, operator precedence and other association rules. *Syntax analysis* is a mapping from source language programs presented in linear concrete notation to abstract syntax trees. This transformation includes lexical analysis, recognition of well-formed source language programs, and association of operators and operands. Semantic interpretation then maps abstract syntax trees to target language programs. Semantic interpretation includes name space control (scope rules), type checking, formal transformations (optimizations), and code generation.

An *abstract grammar* denotes a class of ASTs. The next section shows that every abstract grammar gives rise to an *initial algebra*, the starting point for semantic interpretation.

2. Abstract Syntax and Formal Semantics

This section illustrates how abstract syntax fits into approaches to formal semantics. I discuss how abstract syntax provides a better interface between syntactic and semantic definitions than does concrete (parsing) syntax. The remainder of the section comprises a formal definition of the syntax and semantics of abstract grammar.

2.1. Context-Free Syntax and the Principle of Compositionality

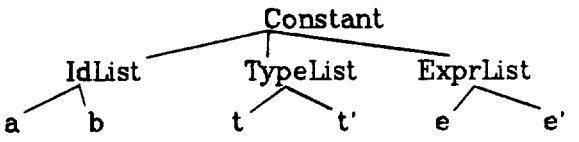
Gottlob Frege's *Principle of Compositionality* states that the meaning of a phrase should be a function of the meanings of its constituent phrases [Dow70a, pg. 8]. The denotational approach to semantic definition requires that the language semantics be consistent with this principle [Sco71a]. The initial-algebra method [Gog77a] requires this as well, since a

semantics is given by a homomorphism from an algebra of terms to a representation algebra whose mathematical properties are well known.

Scott and Strachey do not demand context-free syntax of a programming language. The ADJ group [Gog77a, pg. 76] complain that denotational semantics does depend on context-free syntax. This confusion arises because ADJ is working with a poor interface between the syntax and semantics of languages; they give semantics to parse trees. Scott and Strachey were right; a syntax analyzer need not parse a context-free language to produce an intermediate form, such as ASTs, that allows a homomorphic semantic map. For example, it is well known that a language with a construct such as the following constant declaration is not context-free [Hop79a , pp. 127-128].

const (identifier)^n, (<type–expr>)^n, (<expression>)^n

However, a syntax analyzer could produce an AST such as



for the phrase "const (a)(b), (t)(t'), (e)(e')". Semantics consistent with the principle of compositionality can be given for this AST. The meaning of an IdList, denoted by M⟦IdList⟧, is simply a list of Ids. Similar semantics is given to TypeList and ExprList. The meaning of a const phrase is obtained by transposing the triple

<M⟦IdList⟧, M⟦TypeList⟧, M⟦ExprList⟧>

to obtain a list of triples <Id, Type, Expr>*.

Context-free syntax and semantics that adhere to the principle of compositionality are both desirable, but orthogonal, language features.

2.2. A Formal Definition of Abstract Grammar

Following ADJ's example that gives initial-algebra semantics to context-free (concrete) grammars [Gog77a], this paper gives initial-algebra semantics to abstract grammars. For that we must have a formal description of the syntax of abstract grammar.

2.3. The Syntax

Our example language features various kinds of expressions, among which are addition, subtraction, numeric constants, identifiers and others. These expressions form a *syntactic domain*. Any of the expressions may be used in programs wherever an

expression is required. Types form another syntactic domain comprising identifiers and products. An abstract grammar includes the definition of one or more syntactic domains.

Productions of an abstract grammar describe the form of ASTs. An example of such a production is Add → Expr Expr. This declares that an AST with an Add operator at its root will always have two subtrees, each representing an expression.

The syntax of abstract grammar is defined in a manner similar to that used in formal language theory to define the syntax of context-free grammar [Hop79a, pp. 77-84].

Definition:

An *abstract grammar* is denoted by G = <**SDSymb**, **NonTerm**, **Term**, **SynDom**, **Prod**>.

Let **Var** = **NonTerm** ∪ **Term**.

SDSymb is the set of syntactic domain symbols.

NonTerm is the set of nonterminal symbols.

Term is the set of terminal symbols. (**NonTerm** ∩ **Term** = ϕ)

SynDom : **SDSymb** → $2^{\mathbf{Var}}$

Prod : **NonTerm** → **SDSymb**$^+$

X^+ is the set of sequences $x_1\ x_2\ \ldots\ x_n$, with $x_i \in X$, for $1 \le i \le n$, where $n \ge 1$. Note that the above mentioned sets may be finite or infinite.

Definition:

A tree is an *abstract syntax tree* (AST) for G if all of the following are true:

(1) Every vertex has a label, which is a symbol of **Var**.

(2) If a vertex is interior and has label A, then A ∈ **NonTerm**.

(3) If a vertex is a leaf and has label A, then A ∈ **Term**.

(4) If n has a label A and vertices $n_1, n_2, \ldots, n_k$ are the children of vertex n, in order from left to right, with labels $X_1, X_2, \ldots, X_k$ respectively, then

$$\mathbf{Prod}(A) = Y_1\ Y_2\ \ldots\ Y_k$$

where $X_i \in \mathbf{SynDom}(Y_i)$.

The abstract grammar for our example language is given in Figure 2.1. This grammar describes the ASTs for the sample programs of Figure 1.1, which are displayed in Figure 1.2.

let **Var** = **NonTerm** ∪ **Term**

SDSymb = {Type, Expr, Id, Num} ∪ **Var**

NonTerm = {Constant, Product, Minus, Add, Mult}

Term = Identifier ∪ Number

SynDom (Constant) = {Constant}

SynDom (Product) = {Product}

SynDom (Minus) = {Minus}

SynDom (Add) = {Add}

SynDom (Mult) = {Mult}

SynDom (Type) = {Product} ∪ **SynDom** (Id)

SynDom (Expr) = {Minus, Add, Mult} ∪ **SynDom** (Id) ∪ **SynDom** (Num)

SynDom (Id) = Identifier

SynDom (Num) = Numeral

Prod (Constant) = Id Type Expr

Prod (Product) = Type Type

Prod (Minus) = Expr

Prod (Add) = Expr Expr

Prod (Mult) = Expr Expr

Figure 2.1

Some notational conventions are outlined that simplify the presentation of abstract grammar. This shows the correspondence between the formal notation developed here and the informal notation used in practice. The formal notation, which I will continue to use throughout the paper, allows us to refer to individual components of abstract grammars. The informal notation provides for a finite description of some infinite abstract grammars that would be useful in practice.

Consider, for example, the variable declaration of Pascal. Part of the concrete syntax for this construct is given below. Notice that an arbitrary number of identifiers may be declared within a single declaration. The notational conventions given below will be used in describing the abstract syntax of this construct.

VarDecl → var Identifier IdList : Type

IdList → , Identifier IdList

IdList → λ

Notational conventions for presenting abstract grammars

The following abbreviations are defined.

- Abbreviate **Prod** (N) = w as N → w.
- Abbreviate **SynDom** (A) = S as A = S.

The following conventions allow one to omit redundant notation.

- For any abstract grammar, it is assumed that **SDSymb** ⊇ **Var** and **SynDom** (X) = {X} for all X ∈ **Var**. (where **Var** = **NonTerm** ∪ **Term**).
- Notation such as $\{a_1, \ldots, a_n\}$ ∪ **SynDom** (Id) can be written as $\{a_1, \ldots, a_n$, Identifier} where **SynDom** (Id) = Identifier.
- Explicit definitions for **SDSymb**, **NonTerm**, and **Term** may be omitted.

The following conventions allow one to finitely present some infinite grammars.

- Let $\alpha, \beta \in$ **SDSymb***. The notation $A \rightarrow \alpha\ B^*\ \beta$ represents:
  - An infinite sequence of productions:
    $$A_0 \rightarrow \alpha\ \beta,\ A_1 \rightarrow \alpha\ B\ \beta,\ A_2 \rightarrow \alpha\ B\ B\ \beta, \ldots, A_i \rightarrow \alpha\ B^i\ \beta, \cdots$$
  - A new syntactic domain $S_A = \{A_i \text{ for } i \geq 0\}$
  - For all X ∈ **SDSymb**, if A ∈ **SynDom** (X) then for all $i \geq 0$, $A_i \in$ **SynDom** (X)
- $A \rightarrow \alpha\ B^+\ \beta$ is an abbreviation for $A \rightarrow \alpha\ B\ B^*\ \beta$.

The abstract syntax of a Pascal variable declaration is now simple to describe as shown below.

VarDecl → Identifier$^+$ Type

The abstract grammar of our example language given in Figure 2.1 can be more consisely expressed as displayed below.

Type = {Product, Identifier}

Expr = {Minus, Add, Mult, Identifier, Numeral}

Constant → Identifier Type Expr

Product → Type Type

Minus → Expr

Add → Expr Expr

Mult → Expr Expr

2.4. The Semantics

2.4.1. Initial Algebra Semantics

Before diving into this presentation of semantics, I shall explain some notation and the definitional method used.

A linear notation is used to express trees. Let x () indicate the tree consisting of the singleton (leaf) node labeled x. Let x $(t_1 \cdots t_n)$ signify the tree whose root is labeled x and which has proper subtrees $t_1, \ldots, t_n$ as immediate descendents of the root.

An *initial algebra semantics* is given for abstract grammar [Gog77a]. In this method, source ASTs are given meaning by transformation into target ASTs. Some algebraic preliminaries will ease understanding of this definitional method.

An *S-sorted Σ-algebra*, A, comprises

- S, a set of symbols called the *sorts* of A
- Σ, an $S^* \times S$-indexed family of disjoint sets of operator symbols, called the *operator domain* or *signature* of A. $\Sigma_{w,s}$ in Σ is the set of *operator symbols* of type <w,s>, where $w \in S^*$ and $s \in S$. $\lambda \in S^*$ denotes the empty string
- an S-indexed family of sets called the *carriers* of A. A_s is the carrier of sort $s \in S$.
- an *interpretation* defined as follows. For each $\langle w,s \rangle \in S^* \times S$ and for each $\sigma \in \Sigma_{w,s}$, there is an operation σ_A of type <w,s>, ie: $\sigma_A : A_{w_1} \times A_{w_2} \times \cdots \times A_{w_n} \rightarrow A_s$, where $w = w_1 \cdots w_n$ and $w_i \in S$ for $1 \le i \le n$. An operation σ_A of type $\langle \lambda, s \rangle$ is a constant of sort s ie: $\sigma_A \in A_s$.

By varying S and Σ we get the class of *many-sorted algebras*. By fixing S and Σ we get $\mathbf{Alg}_\Sigma$, the class of *Σ-algebras*.

An algebra, A, is *initial* in a particular class **C** of algebras iff for all B in **C** there is a unique homomorphism h_B:A → B. In any class of algebras there is a unique (up to isomorphism) initial algebra. Later in the paper it is shown that from an abstract grammar, G, for a programming language, L, an operator domain Σ^G can be constructed. With Σ^G, an initial algebra, A, in the class $\mathbf{Alg}_{\Sigma^G}$ can be defined. The semantics of L can be given by specifying a target algebra T in $\mathbf{Alg}_{\Sigma^G}$. Since A is initial, there exists a unique semantic map h_T:A → T, which assigns "meanings" in T to all ASTs of L.

Example of an algebra

Let us define a Σ'-algebra, B. Let the set of sorts be {Type, Expr, ConstDecl, Id}. Assume for simplicity that the identifiers used for types, expressions and constants do not overlap. Let TypeId, ExprId, ConstId be pairwise disjoint subsets of Identifier. I will be able to remove this restriction later. Let Σ' include

$\Sigma'_{\lambda,\ Expr}$ = ExprId ∪ Numeral

$\Sigma'_{Expr,\ Expr}$ = {Minus}

$\Sigma'_{Expr\ Expr,\ Expr}$ = {Add, Mult}

$\Sigma'_{\lambda,\ Type}$ = TypeId

$\Sigma'_{Type\ Type,\ Type}$ = {Product}

$\Sigma'_{\lambda,\ Id}$ = ConstId

$\Sigma'_{Id\ Type\ Expr,\ ConstDecl}$ = {Constant}

Define the carrier B_{Expr} as the smallest set of trees such that

(1) i () ∈ B_{Expr} for all i ∈ ExprId ∪ Numeral

(2) if $t_0, t_1 \in B_{Expr}$ then

Minus $(t_0) \in B_{Expr}$

Add $(t_0, t_1) \in B_{Expr}$

Mult $(t_0, t_1) \in B_{Expr}$

Define the carrier B_{Type} as the smallest set of trees such that

(1) i () ∈ B_{Type}, for all i ∈ TypeId

(2) if $t_0, t_1 \in B_{Type}$ then

Product $(t_0, t_1) \in B_{Type}$.

Define the carrier B_{Id} as the smallest set of trees such that

i () ∈ B_{Id}, for all i ∈ ConstId

Define the carrier $B_{ConstDecl}$ as the smallest set of trees such that

if $t_0 \in B_{Id}$, $t_1 \in B_{Type}$ and $t_2 \in B_{Expr}$ then

Constant $(t_0, t_1, t_2) \in B_{ConstDecl}$.

Typical members of $B_{ConstDecl}$ are displayed in Figure 1.2.

Define the operations

- $i_B = i\ ()$ for all $i \in TypeId \cup ExprId \cup ConstId \cup Numeral$
- if $t_0, t_1 \in B_{Expr}$ and $t_2, t_3 \in B_{Type}$, $t_4 \in B_{Id}$ then

  $Minus_B\ (t_0) = Minus\ (t_0)$

  $Add_B\ (t_0, t_1) = Add\ (t_0, t_1)$

  $Mult_B\ (t_0, t_1) = Mult\ (t_0, t_1)$

  $Product_B\ (t_2, t_3) = Product\ (t_2, t_3)$

  $Constant_B\ (t_4, t_2, t_0) = Constant\ (t_4, t_2, t_0)$

The definition of the Σ'-algebra, B, is now complete.

2.4.2. Constructing an Initial Σ-algebra

One can define an initial Σ-algebra by the following construction [Gog77a]. Let A be an S-sorted Σ-algebra. For each $s \in S$, the carrier A_s is defined as the smallest set of trees such that

(1) if $\sigma \in \Sigma_{\lambda, s}$, then $\sigma\ () \in A_s$

(2) for all $w \in S^+$ ($w = w_1\ w_2 \cdots w_n$ where $w_i \in S$), if $\sigma \in \Sigma_{w,s}$ and $t_i \in A_{w_i}$ $1 \le i \le n$,

then $\sigma\ (t_1 \cdots t_n) \in A_s$

There is an operation σ_A for each $\sigma \in \Sigma$ as follows:

(1) if $\sigma \in \Sigma_{\lambda, s}$, then $\sigma_A = \sigma\ ()$,

(2) for each $\langle w,s \rangle \in S^+ \times S$ if $\sigma \in \Sigma_{w,s}$, then $\sigma_A(t_1, \ldots, t_n) = \sigma\ (t_1 \cdots t_n)$ with $t_i \in A_{w_i}$ $1 \le i \le n$

The proof that a Σ-algebra constructed in this way is initial in $\mathbf{Alg}_\Sigma$ is found in [Bir70a]. One can see that the algebra, B, defined in the example is initial in $\mathbf{Alg}_{\Sigma'}$.

2.4.3. The Initial-Algebra Semantics for Abstract Grammar

Using the initial-algebra construction given above, I show that any abstract grammar gives rise to an initial algebra. If we are to give semantics for practical abstract grammars, we must be able to deal with polymorphic operators. Consider the abstract grammar given below that describes a fragment of the Pascal type system.

Type = {Identifier, SubRange, Array}
SimpleType = {Identifier, SubRange}
SubRange → Expr Expr
Array → SimpleType Type

SubRange is polymorphic; SubRange ∈ **SynDom** (Type) and SubRange ∈ **SynDom** (SimpleType). We can distinguish between two flavors of this operator by affixing superscripts such as

$SubRange^{\text{Expr Expr, SimpleType}}$
$SubRange^{\text{Expr Expr, Type}}$

This technique is used in the construction of an initial-algebra given below. There, each flavor of an abstract grammar operator is considered to be a distinct operator of the algebra.

The treatment of polymorphic operators is an extension of section 3.1 of [Gog77a].

Theorem:

Any abstract grammar gives rise to an initial algebra.

Proof:

Let G = <**SDSymb**, **NonTerm**, **Term**, **SynDom**, **Prod**> be an abstract grammar. We can construct an **SDSymb**-sorted Σ^G-algebra. First define the signature Σ^G as follows

For each $\langle w,s\rangle \in \textbf{SDSymb}^+ \times \textbf{SDSymb}$

$\Sigma^G_{w,s} = \{A^{w,s} \mid A \in \textbf{SynDom}(s) \text{ and } \textbf{Prod}(A) = w\}$

Also, $\Sigma^G_{\lambda,s} = \{A^{\lambda,s} \mid A \in (\textbf{Term} \cap \textbf{SynDom}(s))\}$ for all $s \in \textbf{SDSymb}$

The superscripts are added to the above symbols only to guarantee that the sets $\Sigma_{w,s}$ in Σ are disjoint.

G_s is the carrier of sort $s \in$ **SDSymb**. Define the carriers as follows:

(1) if $A^{\lambda,s} \in \Sigma^G_{\lambda,s}$, then $A^{\lambda,s}() \in G_s$ for all $s \in$ **SDSymb**

(2) for each $\langle w,s \rangle \in$ **SDSymb**$^+ \times$ **SDSymb**

if $A^{w,s} \in \Sigma^G_{w,s}$ and $t_i \in G_{w_i}$ $1 \le i \le n$ where $w = w_1 \cdots w_n$

then $A^{w,s}(t_1 \cdots t_n) \in G_s$

There is an operation σ_G for each $\sigma \in \Sigma^G$ as follows:

(1) $A^G_{\lambda,s} = A^{\lambda,s}()$ for all $s \in$ **SDSymb**

(2) for each $\langle w,s \rangle \in$ **SDSymb**$^+ \times$ **SDSymb**

if $w = w_1 \cdots w_n$ and $t_i \in G_{w_i}$ $1 \le i \le n$

then $A^G_{w,s}(t_1, \ldots, t_n) = A^{w,s}(t_1 \cdots t_n)$

Vertex labels within ASTs of G can be decorated with the superscripts that decorate operators of Σ^G (ie: $A^{w,s}$); such information is available in the AST. Then, G_s comprises ASTs of G having A-labeled roots for $A \in$ **SynDom** (s). G is initial in **Alg**$_{\Sigma^G}$.■

Thus, any abstract grammar, G, is a description of an initial **SDSymb**-sorted Σ^G-algebra, G. For any chosen target algebra, $T \in$ **Alg**$_{\Sigma^G}$, there exists a unique homomorphism h_T: $G \to T$, which assigns "meanings" in T to all ASTs of G.

3. Experience

We have put some of the ideas developed here to practical use. A syntax-analyzer constructor, Sac, has been designed and implemented [Rol82a]. This software tool accepts as input a context-free grammar augmented with simple annotations that describe a parse tree to AST translation. From this input, it produces a compiler component that translates a source language program into an AST.

Experience in maintaining translators built with Sac indicates that changes made to the concrete syntax of the language that do not affect the abstract syntax have no impact upon the semantic interpretation phase.

4. Conclusions

The ADJ group shows how to give semantics to context-free grammars (CFGs). However, nonterminals of a CFG may not be the operators to which one wants to give semantics. A parse tree of a CFG does not necessarily associate operands to operators in a uniform manner. Abstract syntax trees (ASTs) are a better intermediate form for semantic translation.

This paper shows that one can give semantics to abstract grammars. It demonstrates that any abstract grammar gives rise to an initial algebra where ASTs are terms of that algebra. The abstract grammars considered here may include polymorphic operators, which are used in practical abstract syntax.

References

Bir70a. G. Birkhoff and J. D. Lipson, "Heterogeneous Algebras," *J. Combinatorial Theory*, (8) pp. 115-133 (1970).

Dow70a. Dowty, Wall, and Peters, *Introduction to Montague Semantics*, Dordrecht, Boston (1970).

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).

McC63a. J. McCarthy, "Towards a Mathematical Science of Computation," *Congress. Proc. IFIPS Congress 1962*, pp. 21-28 North Holland, (1963).

Pag81a. F. G. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Englewood Cliffs, NJ (1981).

Rol82a. Eugene J. Rollins, "A Syntax Analyzer Constructor," CS/E-82-06, Oregon Graduate Center, Beaverton, Or (August 1982).

Sco71a. D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages," pp. 19-46 in *Computers and Automata*, ed. J. Fox,Wiley, New York (1971).