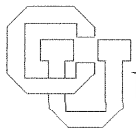
Maptool – Mapping Between Concrete And Abstract Syntaxes

Basim M. Kadhim
William M. Waite

CU-CS-765-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Maptool – Mapping Between Concrete and Abstract Syntaxes

Basim M. Kadhim William M. Waite

CU-CS-765-95 February 1995

University of Colorado at Boulder

Technical Report CU-CS-765-95
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Maptool – Mapping Between Concrete and Abstract Syntaxes

Basim M. Kadhim William M. Waite

February 1995

Abstract

Abstract syntax trees are often used as the basis for performing semantic computations on textual input. In order to build such a tree, there must be a unique correspondence between the abstract syntax and the phrase structure of the input, called the concrete syntax. Although the abstract and concrete syntaxes are often quite similar, expressing semantic computations is often simplified if differences are allowed. This paper will describe a tool, called Maptool, which is designed to aid in the development of the concrete and abstract syntaxes and automatically construct an abstract syntax tree while taking into account differences between the syntaxes.

1 Introduction

The meaning of a construct in a programming language is often described in terms of the meanings of the components of that construct. A tree embodies the relationship between constructs and their components, and therefore much of the analysis that a compiler performs on a source program is conveniently expressed in the form of computations over a tree. Although the structure of this tree is related to the phrase structure of the source program text, the two are generally not identical.

Both the phrase structure and the tree structure can be defined by context-free grammars. The grammar describing the phrase structure is called the *concrete syntax*, while the grammar describing the tree structure is called the *abstract syntax*. Together, the two grammars provide the foundation for verifying the syntactic and semantic correctness of a program in the language.

The concrete syntax can be input to a parser generator, which will then produce a parser for the language. In order for the concrete syntax to be usable by a parser generator, it must be unambiguous and typically must have either the LL(1) or LALR(1) property (depending on the parser generator being used).

The abstract syntax describes the structure of trees over which computations are to be performed. It can be used as the basis of an attribute grammar that describes a particular computation. An attribute grammar is submitted to an attribute evaluator

generator that will produce a program to carry out the computation. Since the abstract syntax describes the structure of trees as opposed to the phrase structure of an input string, it is typically not usable as a concrete syntax. (It may be ambiguous with respect to the input language, and may not conform to the LL(1) or LALR(1) property required of the concrete syntax.)

Each fragment of the tree represents some phrase in the input text, and therefore the correspondence between the concrete syntax and the abstract syntax defines the tree construction process. Tree construction is typically described by manually annotating productions of the concrete syntax with executable code called *semantic actions*. Semantic actions provide an operational specification that is tedious to write and cannot be checked mechanically. *Maptool*, described in this paper, establishes a correspondence between the concrete syntax and the abstract syntax and derives the appropriate semantic actions from that correspondence. Since the correspondence is determined solely from the set of declarative specifications provided by the user, users of Maptool are spared the need to provide and debug an operational specification.

Maptool is a component of the Eli System [3]. Among other things, Eli can take as input a concrete syntax, an attribute grammar, and a mapping specification. The concrete syntax is provided by the user in the form of a context-free grammar written in extended BNF notation. That grammar is translated to strict BNF by an ancillary tool before being input to Maptool. Abstract syntax rules are extracted from the attribute grammar by another ancillary tool. Maptool and both of the ancillary tools are managed by Eli. None of them are visible to the user.

A mapping specification is written by the user in a language designed specifically for Maptool. Maptool allows very specific kinds of differences between the concrete and abstract syntaxes. The mapping specification is designed to provide the additional information needed to allow those differences.

While there are portions of the concrete and abstract syntaxes that differ, there are also many that are identical. In order to minimize redundant effort and reduce the number of places that must be changed if the language is modified, it is useful to be able to omit fragments of one grammar if the information they carry can be deduced from required portions of the other grammar. For example, many phrases do not have any computations associated with them and therefore there is no need to write attribute grammar rules describing the structures of the corresponding tree nodes: Maptool can deduce their abstract syntax from the concrete syntax productions describing the phrase structure. Other phrases *do* have computations associated with them and therefore attribute grammar rules must be provided. If the phrase structure is the same as the structure of the corresponding tree node in these cases, there is no need to include concrete syntax productions because Maptool can deduce them from the abstract syntax describing the tree structure.

The capability for partial specification affords more flexibility in developing new translators because it is not necessary to specify the complete concrete and abstract syntaxes from the outset. For example, a user can start with a complete concrete syntax for an existing language and gradually add attribute grammar rules as needed for tree compu-

```
Program ::= Statement+ .
Statement ::= Computation ';' .
Computation ::= Expr / LetExpr / WhereExpr .

LetExpr ::= 'let' Definitions 'in' Expr .
WhereExpr ::= Expr 'where' Definitions .
Definitions ::= Definition // ',' .
Definition ::= Identifier '=' Expr .

Expr ::= Expr '+' Term / Expr '-' Term / Term .
Term ::= Term '*' Factor / Term '/' Factor / Factor .
Factor ::= '-' Factor / Primary .
Primary ::= Integer / Identifier / '(' Computation ')' .
```

Figure 1: Concrete Syntax

tations. A user could also start with a complete abstract syntax for a new language and then add concrete productions as needed to remove parsing ambiguities.

We begin with an illustrative example showing the use of Maptool, followed by a more complete description of its functional characteristics, and then present some of the interesting aspects of its implementation. A brief history of grammar mapping in Eli and a survey of related work concludes the paper.

2 Example

Consider a language describing simple integer arithmetic expressions with bound identifiers, whose concrete syntax is shown in Figure 1. (The notation `X+` means a sequence of one or more occurrences of `X`, and `X // y` means a sequence of one or more occurrences of `X` separated by `y`. `X*`, meaning a sequence of zero or more occurrences of `X`, and `[X]`, meaning an optional occurrence of `X`, are also allowed.)

The language of Figure 1 allows a sequence of computations terminated with semicolons. Computations can be simple arithmetic expressions defined with the usual rules of precedence or they can be expressions containing bound identifiers. The latter can be specified in two semantically equivalent ways, each consisting of a list of definitions, separated by commas, and an expression.

A `Definition` binds its identifier to the value of the expression specified as part of the definition. Nested definitions are possible. To determine the value of an identifier in an expression, the definitions associated with that expression are checked first. If a binding for the identifier is not found there, definitions that lexically precede the identifier use in enclosing computations are checked. The identifiers bound in each set of definitions must be distinct, and there must be a definition for every use of an identifier.

```
let x = 5 in (let y = 2 in x + y);

(x + y where y = 2) where x = 5;
```

Figure 2: Two Sample Inputs

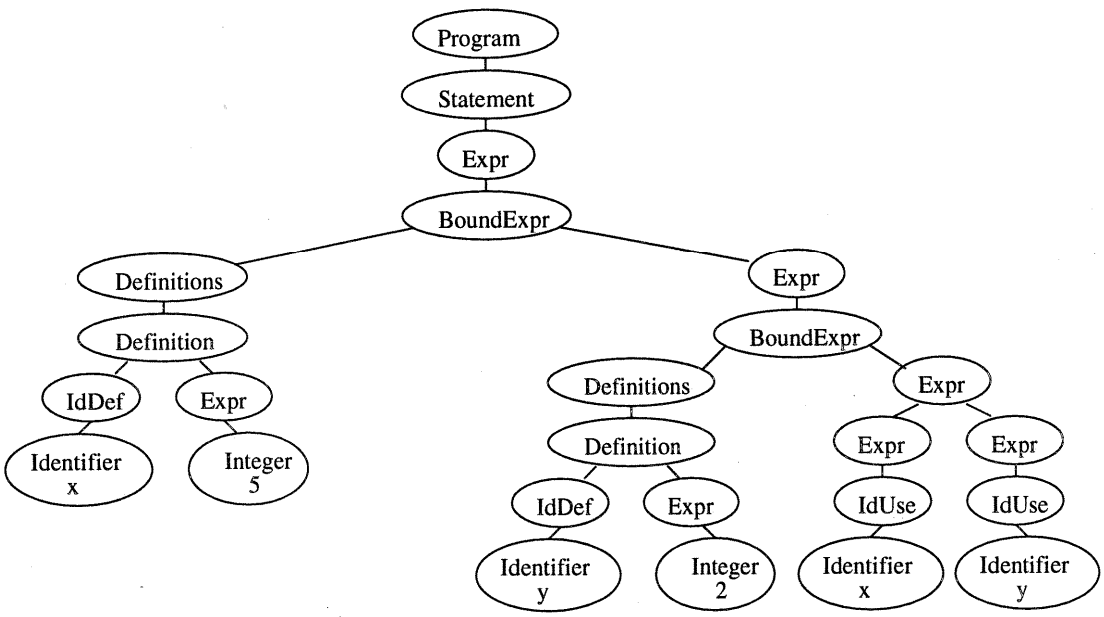


Figure 3: Example Abstract Syntax Tree

We would like to build a processor that validates its input, evaluates the statements, and prints the results one per line. A parser generator can use the concrete syntax to create a parser that will verify the syntactic correctness of the input. Adherence to the binding rules can be verified, and the statements evaluated, by computations over a tree built by the parser. The structure of the tree should facilitate those tasks.

Figure 2 shows two sample inputs to the processor. These two inputs are semantically equivalent, so the parser should build the same tree for either one. In the remainder of this section we will justify the structure of the tree shown in Figure 3, give the abstract syntax that describes it, and indicate the additional information a user must provide to establish the correspondence between the concrete and abstract syntaxes.

In writing new attribute grammar specifications, it is helpful to search for possible decompositions of the problem. In this case, name analysis and expression evaluation are obvious candidates for such a decomposition. Name analysis is responsible for verifying the adherence to the binding rules and for creating the necessary associations between identifier definitions and uses. We begin by analyzing the grammar modifications necessary to support name analysis.

2.1 Name Analysis

If different node types are generated for the `LetExpr` and `WhereExpr` contexts, semantic computations will have to be duplicated because these constructs have identical semantics. This would introduce unproductive redundancy in the attribute grammar specification and make it harder to maintain. Therefore it is reasonable to represent both the `LetExpr` and the `WhereExpr` by the same kind of tree node.

Maptool allows us to define a new symbol, `BoundExpr`, to represent both `LetExpr` and `WhereExpr`. It also allows us to reorder the nonterminals in one of the two concrete productions to match the other. Without this feature, it would be necessary to express the dependencies between the semantic computations slightly differently in each case. With reordering, we can unify the two constructs into a single one and only need to specify the dependencies for the unified construct. It happens that placing `Definitions` first makes the tree computations simpler, so we opt for rearranging the tree nodes in the case of the `WhereExpr` to match the ordering for the `LetExpr`.

To verify adherence to the binding rules of the language, it is also necessary to distinguish between definitions and uses in the abstract syntax tree. This is most conveniently done by using two new symbols, `IdDef` and `IdUse`, instead of the single symbol `Identifier`. `IdDef` will appear in any context where an identifier is being defined, and `IdUse` will appear in any context where an identifier is being used. The fact that `IdDef` and `IdUse` represent specific contexts for the symbol `Identifier` can then be expressed by introducing two productions into the abstract syntax defining them as `Identifier`'s (see Figure 4). Productions such as these, with a single nonterminal on the right hand side, are called *chain rules.*

The first abstract production shown in Figure 4 states that the root node, `Program`, has an arbitrary number of `Statement`'s as children. Similarly, any `Definitions` node has

```
Program LISTOF Statement
BoundExpr ::= Definitions Expr
Definitions LISTOF Definition
Definition ::= IdDef '=' Expr
Primary ::= IdUse
IdDef ::= Identifier
IdUse ::= Identifier
```

Figure 4: Abstract Syntax for Name Analysis

```
MAPSYM
BoundExpr ::= LetExpr WhereExpr .

MAPRULE
LetExpr ::= 'let' Definitions 'in' Expr < $1 $2 > .
WhereExpr ::= Expr 'where' Definitions < $2 $1 > .
```

Figure 5: Mapping Specification for Name Analysis

an arbitrary number of `Definition` children. Notice that the extended BNF definitions for `Program` and `Definitions` in Figure 1 are quite different, but each implies an arbitrary number of children. Use of the `LISTOF` construct allows Maptool to construct a tree that can be traversed efficiently, regardless of the specific phrase structure. The phrase structure corresponding to the `LISTOF` construct may prescribe a specific ordering of the child nodes and may describe literal delimiters.

Comparing Figure 1 with Figure 4, we see that each has productions that have no counterpart in the other. Other productions, like the one for `Definition`, have the same structure in both grammars but differ in some symbol. When a chain rule in the abstract syntax relates a symbol used in the concrete grammar production to the symbol used in the corresponding position of the abstract grammar production, Maptool can establish a correspondence between them without any additional information. There are also productions whose counterparts have different structures and are stated in terms of different symbols. For example, the concrete productions for `LetExpr` and `WhereExpr` both correspond to the abstract production for `BoundExpr`, but there isn't a direct match. Additional information, in the form of a mapping specification, is needed to establish such correspondences.

The mapping specification shown in Figure 5 consists of two parts: one that relates symbols (introduced by the keyword `MAPSYM`), and one that relates rules (introduced by `MAPRULE`). The first part says that the symbol `BoundExpr` is used in the abstract syntax to represent both of the concrete symbols `LetExpr` and `WhereExpr`. The second shows two concrete productions and the ordering of the right hand side nonterminals

```
MAPSYM
Expr ::= Computation Term Factor Primary .
```

Figure 6: Mapping Specification for Expression Evaluation

for the corresponding abstract rule. Although the nonterminals of the `LetExpr` are not reordered, the specification is necessary to indicate that there are no literals in the abstract production. When rule mapping is used, literal symbols are omitted from the specification of the corresponding rule in the abstract syntax, because they are not significant in the matching process. Literal symbols *are* significant otherwise.

It is important to note that would be possible to generate a processor that performs name analysis from an attribute grammar based on the abstract syntax shown in Figure 4. It would not be necessary to specify the remaining portions of the abstract syntax, since they are not relevant to the task of name analysis. This allows for a modular approach to specification development.

2.2 Expression Evaluation

We now want to provide the specifications for evaluating the expressions. We might first consider using the concrete rules for expressions in the abstract syntax without modification. In defining the necessary computations, it becomes apparent that there are a number of chain rules, such as `Expr: Term` that have no semantic significance in the abstract syntax, but nonetheless require us to propagate attribute values through them.

Because no semantic distinction is needed between the symbols `Expr` and `Term`, Maptool allows us to create a symbolic equivalence class for them. `Computation`, `Factor`, and `Primary` belong to this symbolic equivalence class as well and Figure 6 shows the mapping specification that describes it.

Application of this symbolic equivalence class now yields a number of rules of the form `Expr: Expr`. Rules such as this one, in which the left and right hand sides are identical, are called *trivial chain rules.* Since trivial chain rules are not semantically meaningful, they are not included in the abstract syntax and no tree nodes are generated for them. As a result, it is no longer necessary to needlessly propagate attributes through these chain rules in our computations.

The abstract syntax fragment for expressions after introducing the symbolic equivalence class is shown in Figure 7. This abstract syntax fragment is used as the basis for the attribute grammar specification to evaluate the expressions and print the results.

It is important to note that the rule `Primary: IdUse` that appeared in Figure 4 as part of our name analysis specification must be changed to reflect that a `Primary` is now an `Expr` in the abstract syntax.

```
Statement  ::= Expr ';'
Expr ::= Expr '+' Expr
Expr ::= Expr '-' Expr
Expr ::= Expr '*' Expr
Expr ::= Expr '/' Expr
Expr ::= '-' Expr
Expr ::= Integer
Expr ::= IdUse
Expr ::= BoundExpr
```

Figure 7: Abstract Syntax for Expression Evaluation

3 Functional Characteristics

The Eli system hides many of Maptool's interfaces from the user. All that an Eli user needs to know is that Eli accepts concrete grammars, attribute grammars, and mapping specifications as indicated in the last section. From these specifications, it creates a program that parses the specified source text, builds a tree, and carries out the specified computations.

Maptool works closely with LIGA [5], the attribute evaluator generator component of Eli, and also provides input to the scanner and parser generator components. This section gives a more complete and precise description of the interfaces and functional behavior of Maptool, including a discussion of the matching process.

3.1 Interfaces

Maptool accepts three different kinds of specifications. The first is a strict BNF grammar for the concrete syntax. The second is a grammar describing the abstract syntax, also in strict BNF except for productions of the following form:

```
A LISTOF B | ... | D
```

This construct denotes a *variadic* tree node (a tree context in which the left hand side is the parent to any number of children). Each child node is one of the symbols following `LISTOF`.

The third input to Maptool is a mapping specification written in a language designed specifically for Maptool. The details of this specification language are discussed in the next section.

From these inputs, Maptool generates the complete concrete and abstract syntaxes. By "complete", we mean that there are no unreachable symbols: Each grammar has a distinct root symbol from which all other symbols of that grammar can be derived. Since users are allowed to provide partial syntax descriptions, this property may not hold for

the concrete or abstract syntaxes individually in the input. Maptool must, however, be able to deduce complete versions of both after the mapping is complete.

The generated abstract syntax is combined with the user's attribute grammar specifications and provided as input to the LIGA attribute evaluator system. This is because LIGA requires a complete definition of the tree for its processing and the user's attribute grammar specifications alone may not (as in the case of Figure 4) describe a complete tree. While the complete concrete syntax is not used as input to any other tools, it is important for users wanting to debug their specifications or simply to view the complete version of the grammar.

Maptool also generates a parsing grammar. The parsing grammar is the input provided to a parser generator to parse the input language and build the corresponding tree. It consists of the complete concrete syntax, annotated with semantic actions that invoke tree construction functions exported by LIGA. The choice of the tree construction functions is determined by Maptool in its matching process.

3.2 Mapping Specification

A user may specify two kinds of mappings between the concrete and abstract syntax using Maptool. The first kind of mapping defines a symbolic equivalence class. A symbolic equivalence class specifies that certain symbols in the concrete syntax are equivalent from a semantic perspective. These symbols are grouped together and given a single name in the abstract syntax, called their *equivalence symbol.* The concrete symbols that belong to a particular equivalence class must all be terminal symbols or all be nonterminal symbols. This is so that there is no ambiguity in the role of the equivalence symbol in the abstract syntax.

Equivalence classes are useful in almost every set of specifications because distinct symbols for the same concept must be introduced into the concrete syntax to avoid ambiguity in parsing. A very common example of this is in arithmetic expressions, as illustrated in Section 2.

The second kind of mapping establishes correspondences between specific concrete and abstract rules. Such a mapping requires that the corresponding productions have the same number of non-literal symbols on the right hand side. An important advantage that this feature provides is that literal symbols are not significant in instances where rules are mapped explicitly. Concrete rules that differ only in their literal symbols can thus be mapped to the same abstract rule. In addition, when mapping rules it is possible to reorder the nonterminal symbols on the right hand side of the rule. An example of the usefulness of such a mapping for standardizing different input representations was shown in Section 2.

3.3 Rule Matching and Grammar Analysis

Maptool matches rules from the concrete and abstract syntaxes in order to determine which rules from each grammar should be included in the other and to determine how to

correctly generate the annotations used in the parsing grammar. It begins by establishing all of the symbolic equivalence classes. Applying the symbolic equivalence classes to the concrete syntax typically reveals trivial chain productions, in which the left hand side is identical to the right hand side. Section 2 showed an example of this caused by equating symbols used in the concrete syntax for the purpose of establishing the precedence of arithmetic expressions. These trivial chain rules are not included in the abstract syntax.

After the application of symbolic equivalence classes, Maptool matches concrete rules to each `LISTOF` construct in the abstract syntax. This is done by first finding the set of all concrete rules having the same left hand side as that of the `LISTOF` rule. If the set is empty, then there are no concrete rules to match the `LISTOF` rule and a canonical left-recursive expansion of the rule is added to the concrete syntax. If the set is not empty, a recursive matching process is started: For each rule in the set, the right hand side is examined for non-literal symbols that are not equivalent to any of the symbols in the `LISTOF` construct. For each of those symbols, all concrete rules with that left hand side are added to the set. If that symbol is a terminal symbol, an error must be flagged. (This is an indication that a symbol other than an intermediate nonterminal symbol or a symbol from the right hand side of the `LISTOF` construct can be derived from its left hand side.) The recursive matching process continues until no more rules can be added to the set. The resulting set is the set of rules that match the `LISTOF` construct.

Once the matching of lists is complete, Maptool attempts to match each concrete rule that has not already been matched by an explicit mapping specified by the user or by a `LISTOF` construct. The simplest kind of match occurs when the rules specified in the concrete and abstract syntaxes are identical after the symbols of the concrete rule have been mapped to their symbolic equivalents. Another kind of match, involving abstract chain rules, is possible.

A chain rule is a production that has a single symbol on the right hand side. It creates a *link* between two symbols. Linkage is transitive: If a symbol `X` is linked to a symbol `Y`, and `Y` is linked to `Z`, then `X` is also linked to `Z`. If a concrete and an abstract syntax rule differ only in symbols for which a unique link of abstract chain rules exists, then Maptool can deduce a match between those two rules.

In addition to rule matching, Maptool performs some analysis of the grammars. It guarantees that both grammars are reduced, i.e., that there aren't multiple potential start symbols and each nonterminal can generate a sequence of terminals.

Once all matching and analysis is complete, Maptool must determine which concrete rules must be included in the abstract syntax and vice versa. Concrete rules that do not have matching abstract rules are included in the abstract syntax with two exceptions. The first is the trivial chain productions described earlier. The second is a variant of a trivial chain production in which the right hand side has a number of literal symbols in addition to the single non-literal symbol that is equivalent to the left hand side. For example, such a concrete rule might exist to deal with parenthesized expressions:

```
Expr: '(' Expr ')' .
```

These kinds of rules only serve a purpose in the abstract syntax if attribute computations are associated with them. Because of this, if such rules do not already appear in the user's specification of the abstract syntax, they are not introduced.

Abstract rules are added to the concrete syntax if they do not have matching concrete rules. One exception to this is chain rules that do not affect the connectivity of the concrete syntax. These are chain rules that were placed by the user in the abstract syntax to introduce a semantic distinction for a single symbol in the concrete syntax. An example of this was shown in Section 2 with respect to identifier definitions and uses.

4 Implementation

Maptool is described by a set of specifications. Eli uses these specifications to produce a C program, which can then be compiled and executed independent of Eli. The most interesting part of the Maptool implementation is its use of OIL, an operator identification library available in the Eli system. Operator identification is a well understood process, first discussed by Aho and Johnson in 1976 [1, 7]. A more complete description with examples of how the syntax mapping problem is cast in terms of operator identification will be presented in the full paper.

5 History and Related Work

Research in the area of syntax mapping in the Eli system [3] began in 1986 with the introduction of a tool called CAGT described in [2]. Prior to the existence of this tool, users of the Eli system were required to manually attach parsing actions to their concrete syntax in order to build an abstract syntax tree for use by the attribute evaluator. CAGT automated this process by taking complete concrete and abstract syntaxes, matching the rules, and attaching the appropriate semantic actions to the concrete syntax to create a parsing grammar. CAGT also used symbolic equivalence classes in its matching process and allowed the specification of concrete chain rules which would not result in nodes being built in the abstract syntax tree.

In 1992, the tool Genlido was introduced and was used in conjunction with CAGT. The purpose of Genlido was to allow users to only specify the abstract syntax partially, i.e., abstract syntax rules were generated from each concrete rule that had no corresponding abstract rule. Maptool is a superset of the functionality provided by CAGT and Genlido. It allows for partially specified concrete syntaxes, provides more sophisticated matching in the presence of chain rules and list constructs, and allows the reordering of children in the construction of a tree node.

While there are a large number of compiler construction tools publically available, none that we know of provides automated tree construction based only on syntax descriptions.

Automated matching of syntaxes is not an issue in compiler tool sets that have no explicit support for semantic computations on abstract syntax trees.

The Cocktail Compiler Construction Tool Box [4] generates a module for manually constructing an abstract syntax tree from a syntax description. The Purdue Compiler Construction Tool Set [6] goes a step further and integrates a notation for building abstract syntax tree fragments into the specification language of its parser generator. Neither system provides an automated approach to matching the concrete syntax with an abstract syntax intended for use with a tool for specifying semantic computations.

6 Conclusion

Mapping between concrete and abstract syntaxes is necessary when users want to use an abstract syntax tree as the basis for semantic computations on textual input. Many compiler generation systems require that this mapping be done by manually by inserting computations to build the abstract syntax tree.

This paper has presented Maptool, which automates the generation of an abstract syntax tree with a minimum amount of user input while retaining flexibility in deciding the structure of the abstract tree. Our experience with Maptool is that it simplifies translator construction and enhances the modularity and reusability of compiler specifications.

References

[1] Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[2] Ali Bahrami. CAGT – an automated approach to abstract and parsing grammars. Master's thesis, Department of Electrical and Computer Engineering, University of Colorado, 1986.

[3] Robert W. Gray, Vincent P. Heuring, Steve P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[4] Josef Grosch and Helmut Emmelmann. A tool box for compiler construction. In *Lecture Notes in Computer Science, 477*, pages 106–116. Springer-Verlag, 1990.

[5] Uwe Kastens. LIGA: A language independent generator for attribute evaluators. Technical Report Reihe Informatik 63, Universität-GH Paderborn, Fachbereich Mathematik-Informatik, 1989.

[6] T. J. Parr, H. G. Dietz, and W. E. Cohen. *PCCTS Reference Manual.* School of Electrical Engineering, Purdue University, West Lafayette, IN, 1.00 edition, August 1991.

[7] Guido Persch, Georg Winterstein, Manfred Dausmann, and Sophia Drossopoulou. Overloading in preliminary Ada. *SIGPLAN Notices*, 15(11):47–56, November 1980.