An Overview of SORCERER: A Simple Tree-Parser Generator*

Terence J. Parr**

University of Minnesota
Army High Performance Computing Research Center
parrt@acm.org

Abstract. Despite the sophistication of code-generator generators and source-to-source translator generators (such as attribute grammar based tools), programmers often choose to build tree parsers by hand for source translation problems. In many cases, a programmer has a front-end that constructs intermediate form trees and simply wants to traverse the trees and execute a few actions. In this case, the optimal tree walks of code-generator generators and the powerful attribute evaluation schemes of source-to-source translator systems are overkill; programmers would rather avoid the overhead and complexity.
We introduce a freely available tool, SORCERER, that is more suitable for the class of translation problems lying between those solved by code-generator generators and by full source-to-source translator generators. SORCERER generates simple, flexible, top-down, tree parsers that, in contrast to code-generators, may execute actions at any point during a tree walk. SORCERER accepts extended BNF notation, allows predicates to direct the tree walk with semantic and syntactic context information, and does not rely on any particular intermediate form, parser generator, or other pre-existing application.
SORCERER was created to aid in the development of a large application [SOP93] that translates scientific FORTRAN programs. It has also been used for a number of smaller translation problems. For example, the trees depicted graphically in this paper were partially translated from terse, textual descriptions to PostScript[3] by a small SORCERER application.

1 Introduction

The construction of computer language translators and compilers is generally broken down into separate phases such as lexical analysis, syntactic analysis, and translation where the input to the translation phase is an intermediate representation (IR) usually in the form of a set of trees. Many translators are built by hand (in a top-down fashion) while others are described in the meta-language of a translator generator system. It is ironic that most translator generators are compiler code-generator generators, even though most translation problems do not involve compilation. Unfortunately, few practical tools exist for the larger scope of source-to-source translation. UNIX utilities such as `sed` and `awk` provide solutions for some translations, but are limited to matching regular expressions and only on a line-by-line basis.

In this paper, we describe a new translation tool called SORCERER that is well suited to both small and large problems. SORCERER differs from code-generator generators in that no "costs" or "weights" are automatically associated with applying a production and, hence, optimal traversals are not the goal; i.e., the tree grammar must be unambiguous and warnings are generated for nonconformant grammars.

A SORCERER description is essentially an unambiguous grammar (collection of rules) in Extended BNF notation, that describes the structure and content of a user's (IR) trees. The programmer annotates the tree grammar with actions to effect a translation, manipulate a user-defined data structure, or manipulate the IR itself. SORCERER generates a collection of simple and self-contained C or C++

* *To Appear at Int'l Conference on Compiler Construction; April 1994.*
** Partial support for this work has come from the Army Research Office contract number DAAL03-89-C-0038 with the Army High Performance Computing Research Center at the U of MN and the Minnesota Supercomputer Institute.
[3] Postscript is a trademark of Adobe Systems Inc.

functions, one for each tree grammar rule, that recognizes IR subtrees and performs the programmer's actions in the specified sequence. For example, a SORCERER-generated tree recognizer may be embedded in a programmer's C or C++ file via EMACS [Sta86] by simply filtering a tree description through SORCERER with the EMACS `shell-command-on-region` command.

Tree pattern matching is done efficiently in a top-down manner via an $LL(1)$-based[4] parsing strategy augmented with syntactic predicates [PQ94] to resolve non-$LL(1)$ constructs (with selective backtracking) and semantic predicates to specify any context-sensitive tree patterns. Tree traversal speed is linear in the size of the IR unless a non-$LL(1)$ construct is specified—in which case backtracking can be employed selectively to recognize the construct while maintaining near-linear traversal speed.

We begin with a comparison of SORCERER to current translation tools.

2 Previous Work

The automatic translation of source programs has been approached from many different angles whether the target be assembly code or another source language. Programmers have used a number of different approaches including:

- hand-written translators.
- attribute grammar based translator generators [Knu68]; e.g., TXL [CC78], Cornell Program Synthesizer [RTD83], Puma [Gro91], Eli [GHL+92], Ox [Bis92], and COCOL [RM89].
- affix grammar based translator generators [Kos71]; e.g., EAG [Seu93].

A number of narrowly defined restructurers and translators have also been built:

- for FORTRAN (e.g., DELTA [Pad89], KAP and Parafrase [EB91]).
- for assembly code generation (e.g., TWIG [AGT89], IBURG [FHP92], BEG [ESL89], and GCC (RTL) [Sta90]).

Ultimately, the true test of a language tool's usefulness is provided by the vast industrial programmer community. Arguments concerning the relative strengths of unrestricted attribute grammars, affix grammars, S-attributed, and L-attributed grammars [LRS74] are frequently irrelevant because most programmers are not even familiar with the terms. Programmers want to use tools that are flexible, that employ mechanisms they understand, and that generate output that is easily folded into their application. They do not want to be forced into an environment with its own shell, esoteric programming language, or unusual input description language. The resulting applications must also be standalone (i.e., their customers must not be required to install the language tool environment). In such cases, the cost of the tool and the cost per delivered application can be an issue.

As a result, SORCERER uses a common description language, generates extremely simple standalone translators in C or C++ which can be debugged by standard source-level debuggers. SORCERER makes only a single assumption about your application program—that the trees, with node type `AST`, be in child-sibling form with the fields `token`, `right`, and `down` defined. Lastly, SORCERER is free; we only request that users acknowledge us.

Most translation tools used in practice are code-generator generators, which solve a sufficiently different problem from source-to-source translation that the use of different tools is warranted. For example, while code-generator generators handle unambiguous grammar as well as ambiguous grammars, they may not handle unambiguous grammars as efficiently as a tool specifically tuned for fast deterministic parsing. Furthermore, code-generators cannot execute actions at any point during the recognition of a production, typically allow only BNF constructs, and are not designed for input tree manipulations.

The available source-to-source translation systems have a number of limitations that we hope to overcome with SORCERER. The existing tools are either specific to FORTRAN translation (e.g., DELTA), rely on tool-specific languages (e.g., TXL) or on uncommon languages such as Modula-2 (e.g.,

[4] We build top-down parsers with one symbol of lookahead because they are usually sufficient to recognize IRs as IRs are often specifically designed to make translation easy; moreover, recursive-descent parsers provide tremendous semantic flexibility

COCOL, Puma)[5], restrict placement of programmer-specified actions to after recognition of entire tree patterns (e.g., Puma), require that their front-end be used (e.g., COCOL, Cornell Program Synthesizer, EAG, Eli, Ox, Puma, TXL), or are difficult to merge with existing applications (e.g., Eli, Cornell Program Synthesizer, TXL). While all of these attribute grammar based tools are useful (some are extremely good at prototyping translators), their practicality has not been adequately established with large industrial bases.

SORCERER can be considered an **extension** to an existing language rather than a total **replacement** as other tools aspire to be. Consequently, programmers can use SORCERER to perform the well understood, tedious, problem of parsing trees, while not limiting themselves to describing the intended translation problem purely as attribute manipulations. SORCERER does not force the user to use any particular parser generator or intermediate representation. Its application interface is extremely simple and can be linked with almost any application that constructs and manipulates trees.

3 SORCERER as a Programming Language Extension

SORCERER's main strength is that it represents simply a shorthand notation for what programmers typically write by hand. This not only makes SORCERER a simple tool to implement and to understand, but it does not artificially introduce limitations on the programmer. Imagine that you have been given the simple task of translating C language assignments, which have been stored in an IR of the form depicted in Figure 1, to Pascal. Figure 2 provides a function that accepts a tree of this form and prints out the assignment with the Pascal "`:=`" assignment operator (assuming that function `expr` exists elsewhere and prints out an expression in Pascal notation). The exact same functionality may be achieved with a SORCERER description as shown in Figure 3. The SORCERER tree description notation is the same as LISP child-sibling notation with the exception that trees are prefixed with "`#`" to distinguish them from the Extended BNF grouping symbols "`(...)`". Actions are enclosed in European quotes `<<...>>`, terminals begin with upper-case letters, and rules (nonterminals) begin with lower-case letters. The rule definition syntax itself is similar to YACC and consistent with ANTLR [PDC92].

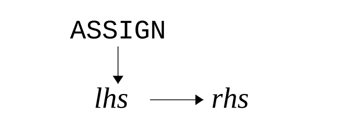
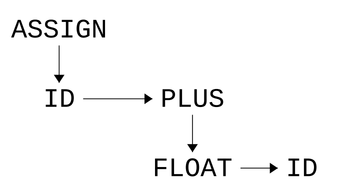
Child-sibling notation	Graphical form
`( ASSIGN` *lhs rhs*`)`	
`( ASSIGN ID (PLUS FLOAT ID))`	

Fig. 1. IR Form for C Assignments

This example illustrates an important feature of SORCERER descriptions—action placement is significant whereas code-generator generators typically have a single action that emits assembly code after the tree pattern is matched. Actions are executed after the recognition of the elements to the left and before the recognition of elements to the right. For example, placing the `printf` action at the end of the `assign` rule would produce incorrect results as it would print the assignment operator after the right-hand-side expression had been printed. Programming languages without an explicit execution order, such as Prolog, are often excellent vehicles for describing problems, but result in programs that are notoriously difficult to debug. Hence, we have chosen to support an implied execution order for SORCERER descriptions.

5 For better or worse, C and C++ are much more prevalent.

```
assign(AST *t)
{
    if ( t->token==ASSIGN ) {
        expr(t->down);
        printf(" := ");
        expr(t->down->right);
    }
    else error;
}
```

Fig. 2. Partial C Code to Translate C Assignments

```
assign  :   #( ASSIGN expr <<printf(" := ");>> expr )
        ;
```

Fig. 3. Partial SORCERER Description to Translate C Assignments

4 SORCERER Description Language

Just as YACC and ANTLR grammars specify a sequence of actions to perform for a given input sentence, SORCERER descriptions specify a sequence of actions to perform for a given input tree. The only difference between a conventional language parser and a tree parser is that tree parsers have to recognize tree structure as well as grammatical structure. For this reason, the only significant difference between ANTLR input and SORCERER input is that SORCERER grammar productions can use an additional grouping construct—a construct to identify the elements and structure of a tree. In this section, we summarize the most important portions of SORCERER syntax.

4.1 Rule and Element Syntax

A SORCERER description is a collection of rules in Extended BNF (EBNF) form and user-defined actions preceded by a header action where the programmer defines the type of a SORCERER tree:

`#header` `<<` *header action*`>>`
actions
rules
actions

where actions are enclosed in European quotes `<<`...`>>` and rules are defined as follows:

rule : $alternative_1$
  | $alternative_2$
  ...
  | $alternative_n$
  ;

Each alternative production is composed of a list of elements where an element can be one of the items in Figure 4. The "..." within the grouping constructs can themselves be lists of alternatives or items from Figure 4. Tree patterns are specified in a LISP-like notation of the form:

```
#( root-item item ... item )
```

where the `#` distinguishes a parenthesized expression from the EBNF grouping construct and `root-item` is a leaf node identifier such as `ID`. Flat trees (lists of items without parents) are of the form:

```
item ... item
```

SORCERER-generated translators can use either user-defined token types or can have SORCERER assign token types to each terminal referenced in the grammar description.

Item	Description	Example
leaf	token type	`ID`
.	wild card	`#( FUNC ID (.)* )`
rule-name	reference to another rule	`expr`
`#(...)`	tree pattern	`#(IF expr slist slist)`
`<<...>>`	user-defined semantic action	`<<printf("%s", t->name);>>`
`(...)`	subrule	`("int" | ID | storage_class)`
`(...)*`	closure	`ID ("," ID)*`
`(...)+`	positive closure	`slist : ( stat | SEMICOLON )+ ;`
`{...}`	optional	`{ ELSE stat }`
`<<...>>?`	semantic predicate	`id : <<is_TYPE(str)>>? ID ;`
`(...)?`	syntactic predicate	`((list EQ)? list EQ list | list)`

Fig. 4. SORCERER Description Elements

4.2 Translation Support

Once the structure of the IR has been described using the notation given in the previous section, actions must be embedded to specify a translation. Actions have access to IR subtrees via node labels and may communicate with other rules via rule parameters and return values analogous to C function parameters and return values; each rule has an implicit parameter `root` which is the subtree to be matched.

Rule elements can be labeled with an identifier, which is automatically defined as an IR tree node pointer, that can be referenced by user actions. The syntax is:

t: *element*

where *t* may be any upper or lower case identifier and *element* is either a token reference, rule reference, or the wild-card. Subtrees may be labeled by labeling the root node. The following grammar fragment illustrates a typical use of an element labels.

```
a   :   #( DO u:ID expr expr #( v:SLIST (stat)* ) )
        <<printf("induction var is %s\n", u->symbol);  analyze_code(v);>>
    ;
```

where `symbol` is some field that the user has defined as part of an `AST`. These labels are pointers to the nodes associated with the referenced element and are only available after the recognition of the associated tree element.

Actions within a rule access trees via labels and communicate via local variables. A special action called an init-action can be specified at the start of any alternative of a rule or subrule. For example,

```
/* match a function call with arbitrary number of arguments */
fc  :   <<int i=0;>>
        #( FUNC ID ( . <<i++;>> )* )
        <<printf("found func call with %d arguments\n", i);>>
    ;
```

Actions may communicate across rules via rule parameters and return values. For example, it is often desirable to pass a value to an expression rule indicating whether it is on the left or right hand side of an assignment:

```
<<enum SIDE { LHS, RHS };>>
stat:   #( ASSIGN expr[LHS] expr[RHS] )
    |   ...
    ;
expr[int side] : ... ;
```

Alternatively, one can pass information between rules as return values. The following example demonstrates how the number of arguments in a function call could be returned and placed into a local variable of the invoking rule.

```
expr:   ID
    |   FLOAT
    |   <<int n;>>
        fc > [n]
        <<printf("func call has %d arguments\n", n);>>
    ;

fc > [int nargs]
    :   <<int i=0;>>
        #( FUNC ID ( . <<i++;>> )* )
        <<nargs = i;>>
    ;
```

4.3 Semantic Predicates

Most source-to-source tree grammars are unambiguous because, unlike code generation, source constructs normally have a single translation in the target language. However, this is not exclusively the case and it is often desirable to specify context-sensitive structures. Many syntactic ambiguities can be resolved and many context-sensitive structures can be expressed easily with *semantic predicates* ([PQ94], [PCD93]). Semantic predicates are user-defined expressions, which evaluate to true or false, that indicate the semantic validity of attempting to match a tree. For example, consider how one would disable certain expression elements when matching the left hand side of an assignment. The following fragment demonstrates how predicates can be used to turn certain alternatives on and off depending on the context.

```
<<enum SIDE { LHS, RHS };>>
stat:   #( ASSIGN expr[LHS] expr[RHS] )
    |   ...
    ;
expr[int side]
    :   <<side!=LHS>>? INT      <<action1>>
    |   ID                      <<action2>>
    ;
```

The SORCERER-generated code to recognize rule `expr` simply incorporates the predicate in the prediction of the first alternative; i.e., the prediction expression would be:

```
( t!=NULL && t->token==INT && side!=LHS )
```

Semantic predicates may also examine the structure or contents of a tree before recognition is attempted by referencing the subtree root, passed implicitly to every rule, called `root`. Figure 5 demonstrates how a predicate can be used to distinguish between the unary and binary operator `Minus`. Because both alternatives begin with a common left-prefix (i.e., `Minus`), SORCERER cannot determine from the root of the tree what follows the `Minus` (how many operands the operator has) with a single symbol of lookahead. The semantic predicate enables rule `expr` to distinguish between the productions.

```
expr:   <<root->down!=NULL && root->down->right!=NULL>>?
        #( Minus expr expr )    // Binary: subtract
    |   #( Minus expr )         // Unary: negate
    ;
```

Fig. 5. Semantic Predicate Used to Isolate Unary Operator

4.4 Syntactic Predicates

Occasionally a programmer is faced with a grammatical construct that cannot be recognized using SORCERER's deterministic $LL(1)$-based approach. Consequently, SORCERER supports the specification of semantic and *syntactic predicates* ([PQ94], [PCD93]) to indicate the semantic and syntactic validity of attempting to match a particular rule alternative, respectively. Figure 5 used a semantic predicate to test the structure of the current input subtree. While this is effective, the semantic predicate is not using semantics (e.g., symbol table information) to resolve the parsing nondeterminism. Syntactic predicates provide a more convenient means of indicating that a particular tree structure is necessary for successful recognition. A more natural way to describe rule `expr` in Figure 5 would be as depicted in Figure 6. The syntactic predicates indicates that to successfully match the binary version of `Minus`, the current subtree must have `Minus` at the root and have two children.

```
expr:   ( #(Minus . .) )?  #( Minus expr expr )
    |                      #( Minus expr )
    ;
```

Fig. 6. Syntactic Predicate Used to Isolate Unary Operator

4.5 Example

In order to illustrate the notation provided in the previous sections, a useful, but small example is presented in this section. We have chosen to describe an interpreter such as one might construct for a small, application-specific language; while it could just as easily have been constructed in C or Pascal, the example illustrates a complete SORCERER description. Consider a trivial LISP-like language which allows the definition of new functions, the application of functions, and the setting of symbols to particular values; the only predefined function is "+". Figure 7 a simple program that computes the value `45` and Figure 8 gives a SORCERER program that will interpret trees of the given form; for clarity and brevity, we have substituted algorithmic code in italics for the actual C++ actions.

The nodes in the IR tree represent the token type stored in field `token` of `AST`. Other information is stored in an `AST` node such as the text of the identifier, `id`, but SORCERER-generated translators use only the structure of the tree and the value in `token` to match IR patterns; fields `right` and `down` are used to navigate the IR.

5 Conclusions

SORCERER was developed out of frustration while trying to solve "real-world" translation problems in the realm of parallel supercomputing. It has since proven valuable for a wide range of smaller problems as well. SORCERER's strength lies primarily in its flexibility and simplicity; because of the tremendous success of ANTLR in the programmer community (well over 1000 registered users in 37 countries), a number of ANTLR features were carried over to SORCERER. Programmers may view translation descriptions as tree-pattern recognition extensions to C or C++. SORCERER-generator translators may be used alone or by folding them into existing applications. SORCERER may be obtained via anonymous `ftp` at `marvin.ecn.purdue.edu` in `pub/pccts/sorcerer`.

6 Acknowledgements

The SORCERER description language was formed through numerous discussions with Aaron Sawdey, Gary Funck, Peter Dahl, Steve Anderson, and Matt O'Keefe at the University of Minnesota. Additionally, the following individuals were instrumental in testing or discussing the SORCERER prototype: Devin Hooker at Ellery Systems, Kenneth D. Weinert at Information Handling Services, and Marlin

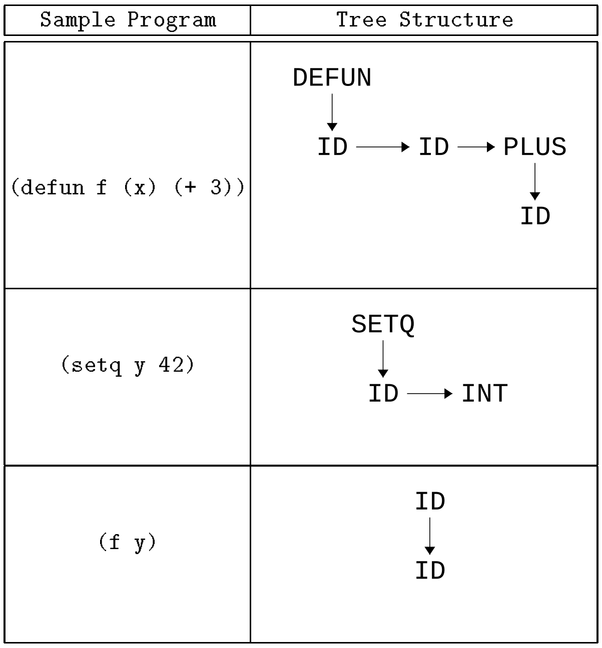


Fig. 7. Sample Program in LISP-Like Language With IR Structure

Prowell at Janus Computing, Anthony Green at Visible Decisions Inc., and John Hall at Worcester Polytechnic Institute.

Gary Funck at Intrepid Technology, Inc. and Aaron Sawdey deserve special recognition for their extensive help with 1.00B SORCERER design and testing.

```
#header <<
#define _PARSER_VARS char *id; int ival; STree *args, *code;
#include "mytokens.h"         // definition of token types; i.e. ID, PLUS, ...
>>

<<
main()
{
    STreeParser p;
    AST *result;
    STreeParserInit(&p);
    /*  a simple ANTLR grammar embodied make-IR-tree-for() in actual test */
    stat1 =  make-IR-tree-for("(defun f (x) (+ 3))");
    stat2 =  make-IR-tree-for("(setq y 42)");
    stat3 =  make-IR-tree-for("(f y)");
    sexpr(&p, &stat1); sexpr(&p, &stat2); result = sexpr(&p, &stat3);
}
>>

sexpr > [AST *result]
    :   <<AST *arg>>
        #( SETQ t:ID sexpr > [arg] )
        << add  t->id to symbol table as  ID and set its value to  arg;  result = t;>>

    |   /*  Don't eval args or code block; hence, we use a wild-card rather than ref to  sexpr */
        #( DEFUN t:ID args:. code:. )
        << add  t->id to symbol table as  DEFUN;>>
        << add  args and  code of func to sym entry;  result = t;>>

    |   <<AST *op1,*op2;>>
        #( PLUS sexpr > [op1] sexpr > [op2] )
        <<result =  make-node-out-of-token-and-value(INT, op1->ival + op2->ival);>>

    |   <<AST *arg, *fval;>>
        <<isfunc(root->id)>>?    /* is ID a function (DEFUN)? */
        #(   ID
             << create a new temporary scope;>>
             << add arg names of function to symbol table;>>
             ( sexpr > [arg] << set arg name of function to evaluated tree:  arg;>>
             )*
             <<fval = sexpr( ptr_to_code_of_function);>>
             << remove all function arg names by removing temporary scope;>>
             << switch to previous scope;>>
        )
        << return  fval;>>

    |   t:ID    <<result = ival  of symbol table entry of  t->id;>>

    |   t:INT   <<result = t;>>
    ;
```

Fig. 8. SORCERER Interpreter For LISP-Like Language

References

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[Bis92] Kurt M. Bischoff. User Manual for Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C. Technical Report TR92-30, Computer Science Department, Iowa State University, December 1992.

[CC78] James R. Cordy and Ian H. Carmichael. *The TXL Programming Language Syntax and Informal Semantics: Version 7.* Software Technology Laboratory; Queen's University at Kingston; Kingston, Canada, 1978.

[EB91] R. Eigenmann and W. Blume. An Effectiveness Study of Parallelizing Compiler Techniques. In *Proceedings of 1991 International Conference on Parallel Processing*, volume II, pages 17–25, August 1991.

[ESL89] H. Emmelmann, F. W. Schröer, and R. Landwehr. BEG — a generator for efficient back ends. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, volume 24 (7), pages 227–237, July 1989.

[FHP92] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992. this was actually published in 1993 due to publications delays.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35:121–131, February 1992.

[Gro91] Josef Grosch. Puma — A Generator for the Transformation of Attributed Trees. Technical Report 26, Gesellschaft für Mathematik und Datenverarbeitung mbH; Universität Karlsruhe, November 1991.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Kos71] C. H. A. Koster. Affix Grammars. *ALGOL 68 Implemenation*, 1971.

[LRS74] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *Journal of Computer and System Sciences*, 9:279–307, 1974.

[Pad89] David A. Padua. The Delta Program Manipulation — Preliminary Design. Technical Report CSRD 808, University of Illinois at Urbana-Champaign, 1989.

[PCD93] Terence Parr, Will Cohen, and Hank Dietz. The Purdue Compiler Construction Tool Set: Version 1.10 Release Notes. Technical Report Preprint No. 93-088, Army High Performance Computing Research Center, August 1993.

[PDC92] T.J. Parr, H.G. Dietz, and W.E. Cohen. PCCTS 1.00: The Purdue Compiler Construction Tool Set. *SIGPLAN Notices*, 27(2):88–165, February 1992.

[PQ94] Terence J. Parr and Russell W. Quong. Adding Semantic and Syntactic Predicates to $LL(k)$—*pred-*$LL(k)$. In *To Appear in Proceedings of the International Conference on Compiler Construction; Edinburgh, Scotland*, April 1994.

[RM89] Rechenberg and Mossenbock. *A Compiler Generator for Microcomputers.* Prentice Hall, 1989.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[Seu93] M. Seutter. Informal introduction to the Extended Affix Grammar formalism and its compiler. Technical Report TR93-19, University of Nijmegen, Netherlands, September 1993.

[SOP93] Aaron Sawdey, Matthew O'Keefe, and Terence Parr. The Translation of Fortran-P to CM Fortran Using Sorcerer. Technical Report Preprint No. 93-102, Army High Performance Computing Research Center, University of Minnesota, October 1993.

[Sta86] R. M. Stallman. *GNU Emacs Manual, Fifth Edition, Emacs Version 18 for Unix Users.* Free Software Foundation, October 1986.

[Sta90] R. M. Stallman. *Using and Porting GNU CC for version 2.0.* Free Software Foundation, November 1990.

This article was processed using the LaTeX macro package with LLNCS style