A*: A Language for Implementing Language Processors

David A. Ladd and J. Christopher Ramming

Abstract—**A* is an experimental language designed to facilitate the creation of language-processing tools. It is analogous either to an interpreted `yacc` with Awk as its statement language, or to a version of Awk which processes programs rather than records. A* offers two principal advantages over the combination of `lex`, `yacc`, and C: a high-level interpreted base language and built-in parse tree construction. A* programmers are thus able to accomplish many useful tasks with little code. This paper describes the motivation for A*, its design, and its evolution. Experience with A* is described, and then the paper concludes with an analysis of that experience.**

Index Terms—**Compilers, application-oriented languages.**

I. Introduction

COMPUTER programs contain a great deal of useful information; often they need to be analyzed both by humans and other computer programs. In Lisp, one early programming language, meta-programs (programs that analyze programs) are easy to write because code and data have the same representation. But from the perspective of a tool builder, programming languages have become increasingly unwieldy over time because the convenience of humans has taken precedence over that of machines. Today, tremendous expertise is required to build tools for many popular languages such as C++ and ADA, and few programmers are able to do so.

Two factors that affect the effort needed to construct a language tool are the syntactic and semantic complexity of the subject language. Syntax issues are the least burdensome: formal language theory is considered part of the core computer science curriculum and has been deeply studied. Consequently, formal syntax specification is common, and tools such as `yacc` which are both practical and efficient are in nearly universal use. By contrast, few languages are defined with a formal semantics, and *ad-hoc methods* are typically used to implement whatever semantics are assigned to a given language.

The extent to which a tool treats the full syntax or semantics of the subject language is another factor in tool construction; many tools are not sensitive to certain details of the language. For instance, a complexity metric for C programs might ignore most declarations and expressions, concerning itself primarily with control flow constructs. But the startup cost for such a metric tool is nonetheless high; a full-blown C parser is the *ante*.

If there were little demand for language tools each one could be developed and maintained independently, but new tools (even for old languages) are frequently being created. Furthermore, new languages continue to appear, particularly those that are application-specific. A particularly compelling use for new languages is to encapsulate some expertise or a solution to a particular problem. As an example, consider the `yacc` tool itself, in which the input language (Backus-Naur form) is used to specify a particular computation (an LR(1) parser). Such languages, designed with a single task in mind, offer flexibility, re-use, and occasionally some formal verification made possible by a restricted domain. But in general an application language approach to software engineering is hard to pursue because it is difficult to write the host of supporting tools (compiler, debugger, pretty-printer, and so on) needed by each new language.

While there are a number of systems that will help one construct full-blown metaprograms such as compilers and interpreters, we wanted something with extremely low overhead. We set out to build a something with the property that it would help even inexperienced users build simple meta-programs in a matter of minutes with a few lines of code. A* is the result; it is more than anything else an engineering exercise, as most of its ideas are not new. It is the arrangement of these ideas and the purpose to which they are directed distinguish A* from other tools.

II. Other Work

Many systems exist for building language tools. Some, such as `yacc` [1] and `lex` [2], solve a particular piece of the overall language tool construction problem. Other tools offer a complete solution for a somewhat narrower class of problem. Two important classes of language tool that have been recognized and addressed by tool-building systems are *language translation* tools and *language analysis* tools.

GENOA [3] is one example of a language analysis tool: the essential idea is to provide a query language that allows users to answer certain questions about arbitrary programs. The GENOA query language is carefully designed, and both the time and space complexity of GENOA queries can be determined statically. But GENOA queries have limited power, and program-to-program transformations (queries that yield programs as their answer) and interpreters (queries that, given an arbitrary program and some input, yield an output) are beyond its scope.

Another class of compiler tool is the translation tool, e.g., a system designed primarily to produce output based on the structure of the input. While there are some commercial systems such as METATOOL [4] designed to offer support for

Manuscript received September 1993; revised March 1995.
The authors are with AT&T Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566; e-mail: {ladd; jcr}@research.att.com.
IEEECS Log Number S95045.

template-drive translation, we suspected that unique notations, administrative overhead (including a basis in compilation rather than interpretation), and the need to address parsing explicitly was already too high a barrier for most people. Another variety of translation system based on tree-transformations (e.g., TXL [5], [6]) is elegant and attractive but of limited value when considering languages such as C that do not have the necessary context-free grammar; in addition, TXL transformations (being homomorphic) restrict its range of applications.

A* was designed to accomodate both transformations and queries equally, is interpreted, has low overhead for simple programs, and is an extension of Awk, a well-known and popular language.

III. Historical Context

Since A* was developed in response to specific software production needs, it seems worthwhile to review the three specific problems which motivated A*.

In part, A* had its roots in an idle wish to build tools for C programs as easily as Awk [7] programmers build tools for text streams. At one time several new C tools were being advertised that we thought were unnecessarily tedious to implement; some were built by modifying the C compiler (effective reuse, if somewhat time-consuming) and others by approximating C's grammar rather than treating it precisely. Modifying the C compiler requires more sophistication and effort than some tools seemed to justify, and any approach using an imprecise grammar has inherent limitations. We thought a better solution might be to invent a language specifically for manipulating C programs.

At approximately the same time, a language called "PRL5" [8] (the Population Rule Language, version 5) was designed. PRL5 is a database constraint language designed to express rules about relational databases. It is a declarative specification language with certain useful formal properties. Because PRL5 is multi-purpose (as are many specification languages), it was clear from the start that many tools to process PRL5 programs would be needed. It seemed that the same technology implied by an "Awk-for-C" might also be useful for rapid PRL5 tool prototyping.

Finally, someone observed that the design of PRL5 should depend on how PRL4 [9] had been used. PRL4 is a complex language, and many of its constructs had rarely or never been used. It became clear that a number of throw-away language tools would help in analyzing the corpus of PRL4 programs.

IV. Goals

The driving problems implied two broad goals: to support the rapid prototyping of language tools and to make the creation of simple "throw-away" language tools feasible. These goals would be supported by several properties:

- A rapid edit-test cycle. We wanted to support rapid prototyping, so a nimble interpreter seemed a more appropriate basis for our tool than a presumably slower compiler.
- High-level statement language. We wanted a language with powerful primitives and high-level facilities tailored for building language tools.
- Low intellectual overhead. Building language tools typically requires much expertise; we hoped for a language that would be easy to learn and that would hide the complexity of language-processing tasks such as parsing.

Many of these properties are commonly associated with Awk, so we began thinking in terms of Awk-for-C and Awk-for-PRL even though there was no preconceived commitment to Awk itself. Nonetheless, these terms seemed descriptive, and the abbreviated form "A*" (A-star) quickly became entrenched. The asterisk is a placeholder for the subject language of one's choice.

V. The Awk Connection

Awk is a language for file processing; the essence of its convenience is a control structure implicit in all Awk programs. Awk breaks each input file into records, and each record into fields. The typical Awk program consists of an implicit loop over all records, and within this loop the user's code is executed. The user's program is expressed as a series of "pattern-action" statements. A pattern is a boolean expression, typically based on the value of the current input record or its fields. An action is a statement to be executed whenever the pattern evaluates to "true."

Interestingly, many language processors have a similar structure. An input file is parsed according to a grammar and the resulting parse tree is traversed. At each step in the traversal, the "current node" is examined; depending on its nature, different actions are performed.

Thus there is a striking similarity between the skeleton programs described in Fig. 1 and Fig. 2. Parsing, traversal, and pattern-action processing are present in both. But whereas Awk records are easily parsed, programs have a richer grammar. In addition, traversing a parse tree is likely to be more complicated than traversing a file of records.

```
parse the file into records
for each record in the input file
  parse the record into fields
  for each pattern-action statement
    if pattern is true then
      execute the action
```

Fig. 1. Awk skeleton.

Awk also has a high-level statement language with features that are useful for language processing. For instance, associative arrays are particularly useful for implementing symbol tables, and it appeared initially that automatic variable declaration and initialization supported the goal of rapid prototyping.

Owing to the above observations, A* came to be based on Awk. Its essence was achieved by:

- retaining the Awk statement language and its interpreter;
- providing a mechanism for replacing Awk's parser with

an arbitrary LALR(1) parser;
- providing a new data structure and notation for parse trees;
- providing a way to describe parse tree traversal;
- augmenting the statement language to ease the construction of larger programs.

```
parse the program into a parse tree
for each node in the parse tree
  for each grammar construct
    if the node matches the construct then
      perform a particular action
```

Fig. 2. Typical language-processing skeleton.

VI. The Yacc Connection

Our first priority was to find a scheme for incorporating arbitrary new parsers into Awk. yacc is a *de-facto* standard for language development, and a large number of interesting subject languages—less the yacc language itself, oddly—have been described with yacc. But these descriptions suffer from three problems: first, they have embedded "semantic actions" specific to a tool and irrelevant to the language *per se*; second, a yacc specification without a lexical analyzer is not a complete description of a language; finally, while yacc builds parsers for context-free languages, some users of yacc execute semantic actions during parsing to introduce context-sensitivity. Nonetheless, yacc grammars tend to be the best language descriptions around.

In order to make use of yacc language descriptions, we use a yacc-to-yacc translator that replaces semantic actions with code for building a standardized parse tree usable by A*; the result is then linked with a suitable lexical analyzer and the extended Awk interpreter.

Unfortunately, we were unable to similarly re-use existing lexical analyzers. This is in part because a lex-to-lex translation tool would not be widely applicable since lex is not as popular as yacc. Furthermore, the semantic actions in lex programs are often "impure" in the sense that they both affect the analysis itself and also construct token representations to pass to a parser; automatically separating these two kinds of activity in arbitrary C fragments is generally impossible.

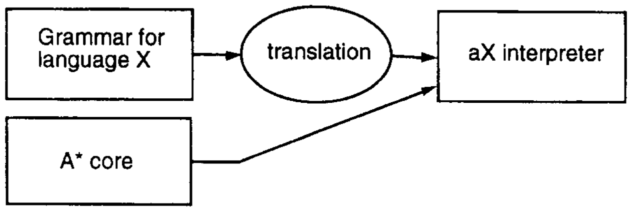

Fig. 3. Building A*.

Fig. 3 illustrates the instantiation of A* for a particular subject language. The resulting interpreter is named by prefixing the subject language name with an "a," thus **ac** is A*-for-C, **aawk** for Awk, **aprl5** for PRL5, etc.

VII. The Language A*

This section will describe A* in general terms, concentrating on rationale and substance rather than details.

A. A*'s Use of Awk

Because A* is based on Awk, it inherits the Awk statement language and, at a stroke, many of its advantages and disadvantages. Among its advantages are that it is interpreted, easy to learn, and "higher-level" than C. On the other hand, serious bugs in Awk programs are frequently observable only at run-time, and Awk programs do not scale well.

B. Parse Trees and Attributes

A* operates by building a parse tree and making that tree available to the user. Because it is necessary to refer to this data structure, syntax was added to the Awk statement language to support references to parts of a tree. A* has a notion of the "current node,"; the notation for the current node is a period (.). Certain pre-defined attributes are computed for each node and these attributes can be dereferenced with the arrow operator; for instance, `.->text` refers to the text associated with the current node. Parse tree nodes may have children, and the array-valued attribute `kids` contains the immediate descendants of a node (thus `.->kids[1]` refers to the first child of the current node). Later, we decided to offer a new form of subscripting that would be symbolic rather than positional: `.->kids[`*`nodetype`*`#`*`n`*`]` now refers to the *n*th child of type *nodetype*. User-defined attributes are also permitted by A*; they are declared at the beginning of a program.

In A* the double dollar-sign (`$$`) can be used as syntactic sugar for `.->`. Originally, A* used `$$0` to refer to the current node and `$$1` through `$$`*`n`* to refer to the children of the current node. This notation was intended to suggest that nodes are analogous to Awk's records and fields. Recall that in Awk, `$0` is the current record and `$1` through `$`*`n`* refer to the fields of the current record. However, we eventually decided that this notation did not co-exist well with the notion of attributes, since a misspelled attribute name would be interpreted as a child node reference with a new variable and silently produce strange results.

C. Tree Surgery

A* produces concrete parse trees, but concrete parse trees are not always the most convenient input representations. To remedy this problem a number of tree surgery primitives were added to A*. These new built-in functions permit the creation of new trees as well as the modification, copying, and destruction of existing trees. One frequent use of these primitives is "tree flattening," in which left-recursive lists in the parse tree are changed from binary trees of height *n* to a single node with *n* children. Fig. 4 illustrates a pattern-action rule for tree flattening. The figure also illustrates an A* convention that grammar symbol names are prefixed with an underscore to avoid confusing these symbols with ordinary variables.

D. Grammatical Patterns

It quickly became apparent that Awk's pattern language is cumbersome for describing parse tree nodes; complicated patterns like the one in Fig. 5 (which identifies an addition node) identify a particular nonterminal by examining its type and the types of its children.

```
$$type==_thing_list {
  if ($$alternation!=1) {
    attach($$kids[1], detach($$kids[2]))
    setdot($$kids[1])
    freenode($$parent)
  }
}
```

Fig. 4. Tree flattening.

```
$$type==_expression &&
$$kids[1]->type==_expression &&
$$kids[2]->type=='+' &&
$$kids[3]->type==_expression &&
$$kids->nc==3 {
    printf "Add("
    traverse($$kids[1])
    printf ","
    traverse($$kids[3])
    printf ")"
}
```

Fig. 5. Infix-to-prefix translation with an Awk-like pattern.

```
_expression : _expression '+'_expression
  {
    printf "Add("
    traverse($$kids[_expression#1])
    printf ","
    traverse($$kids[_expression#2])
    printf ")"
  }
```

Fig. 6. Infix-to-prefix translation with a yacc-like pattern.

But yacc's BNF seemed more compact, so we created a shorthand resembling a yacc grammar rule. Fig. 6 shows the pattern of Fig. 5 expressed in the new format.

E. Traversal Specification

File processing in Awk is simpler than language processing in A*. File traversal is straightforward: one almost always starts at the beginning of the file and works line by line to the end. Parse tree traversal is much more complicated—sometimes preorder traversals are desired, sometimes inorder, sometimes postorder, and sometimes a totally irregular traversal is called for. We needed the flexibility to describe any kind of traversal, and we wanted to preserve a separation between traversal specification and the substance of processing.

The mechanism we settled on was a pair of mutually recursive meta-functions `traverse(`*node*`)` and `dispatch()`. A processing pass over the parse tree is syntactically divided into two parts, the *dispatch section* and the *traversal section*. The dispatch section is analogous to the whole of an Awk program and contains the usual set of pattern-action statements. The traversal section uses the `traverse` and `dispatch` meta-functions to specify the order in which the parse tree should be visited. `traverse(`*node*`)` stacks the current node, makes its argument the new current node, invokes the traverse section, and restores the old current node. `dispatch()` simply invokes the pattern-action sections in the dispatch section.

This scheme for managing tree traversal allows the most common traversals to be specified easily. In practice, almost all traversal sections use the same traversal rule for each node in the tree. Furthermore, that action is most often one of the built-in traversal methods—`preorder()` or `postorder()`—or simply a `dispatch()` back to the other pattern-action section. Some programs have benefited from the generality offered. The register allocator of the PRL5 compiler consists of a complicated traversal section that determines the order in which to walk the tree together with a dispatch section that simply applies local rules at each node.

F. Multiple Passes

Because not all programs can accomplish their purpose with a single pass over the parse tree, notation was introduced to describe multiple passes. Each pass is composed of both a dispatch and traversal section, separated by a double percent sign; in addition, `%{` and `%}` delimit the pass description. A skeleton pass description is illustrated in Fig. 7; any number of these can be concatenated.

```
%{
  // dispatch section
  pattern1 action1
  pattern2 action2
  ...
  patternN actionN
%%
  // traversal section
  pattern action
  ...
%}
```

Fig. 7. A pass description.

We decided that any pattern-action rules preceding the first pass would be interpreted as actions to apply during parsing. Until this reduction-time activity was introduced, one optional difference between A* and yacc was that A* constructed the entire parse tree before traversing it, whereas yacc performed activity only during parsing.

G. Combinations of Printing and Traversal

While at first it seemed desirable to separate traversal control from per-node processing, we found that this was often inconvenient. Many programs were being constructed with an idiomatic interleaving of traversal and printing. Because of this, `printf` was extended with new format characters that allowed traversal of parse tree nodes to be interspersed with the printing of characters. Still more format characters were added to support a consistent indentation style in an output stream. Table I lists the new formatting specifiers.

Fig. 8 uses the new `printf` formats to express the rule shown in Fig. 5.

H. Extragrammatical Support

Few languages are defined completely by a yacc parser; whitespace and comments are typically stripped away by the lexical analyzer and do not become part of the parse tree. But some language processing tools (like pretty-printers) need to treat comments. To accommodate those tool-writers who need to consider such extra-grammatical elements without complicating matters for those who do not, we decided to make comments part of a doubly-linked list of terminals at the fringe of the parse tree. The previous and next tokens are made available through the `prev` and `next` attributes of terminal nodes. Thus, comments are available to programs prepared to traverse this list, and invisible to programs not concerned with them. Fig. 9 shows a fragment of an A* parse tree with a comment linked into the token list.

TABLE I
`printf` FORMATS

`%I`	increase indentation level
`%Z`	decrease indentation level
`%N`	emit a newline and indent
`%T`	traverse the corresponding argument
`%nT`	traverse child n, e.g., `$$kids[n]`

```
 ... // some language constructs
_expression::
  _expression '+' expression
    { printf "Add(%1T,%2T)" }
 ... // all other constructs
```

Fig. 8. Infix-to-prefix translation with new `printf` formats.

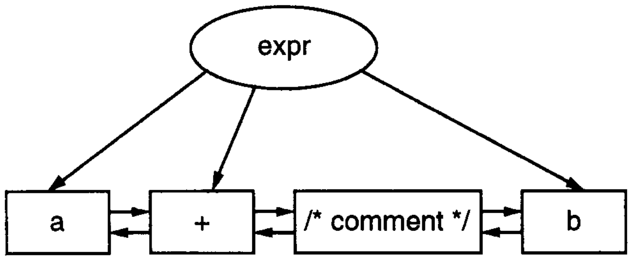

Fig. 9. Linked terminals and comments

I. Improvements to Vanilla Awk

A* also extended the Awk language with certain general improvements. One of these was a local declaration, providing a way to limit the scope of a variable to the enclosing compound statement. Given this syntax for defining local variables, we were able to outlaw a dangerous Awk idiom for limiting the scope of a variable to a function (in Awk, it is not an error to call a function with fewer actuals than formals: the extra formals are initialized to the empty string and can be used as local variables). A* functions are required to have the same number of formal and actual arguments, unless defined with the `varargs` keyword.

Another particularly useful extension was a `switch` statement, with somewhat different semantics from C's—cases are not required to be constant, and to avoid some tedious error handling, any switch not resulting in a match yields a fatal error. Also, unlike C, there is no fall-through from one case branch to the next in the A* `switch` statement.

One further disadvantage of Awk is that no debuggers have been developed for the language. But since we often use debuggers only to show the call stack at the point of a fatal error, we were able to achieve much of the value of a debugger by offering such stack traces as a matter of course when exiting abnormally.

VIII. A* Programming Style

The use of A* evolved over time; one interesting result was the development of two radically different programming styles. Some programs treated only a subset of the subject language and did not require complex traversal specifications. For instance, Fig. 10 displays one version of a complexity metric for C programs. This version computes its result by traversing the implicitly-built parse tree; a shorter version of this program does its work entirely during parsing. Note that calls to `traverse()` and `dispatch()` are localized in the second section, effectively separating traversal specification (in this case preorder) from the pattern-action statements, which simply count the instances of several C keywords.

```
%{
$$type==_SWITCH { DE -= 1}
$$type==_IF ||
$$type==_FOR || $$type==_WHILE ||
$$type==_DEFAULT || $$type==_CASE {DE++ }
$$type==_OR_OP || $$type==_AND_OP { ODE++ }
END {
  print "Simple complexity: "DE + 1"
  print "Full complexity: "DE+ODE+1
}
%%
{ local n // specify a preorder traversal
  dispatch( )
  for (node n in .) traverse(n)
}
%}
```

Fig. 10. A complexity metric program for C.

However, for some tools the action at each node is most naturally expressed as an interleaving of subtree traversal and computation. For such programs the practice of separating traversal control from pattern-action statements is inconvenient. To achieve fine control over traversal in A*, the best solution is a degenerate traversal section which simply calls the meta-function `dispatch()`; `traverse()` is then called from the dispatch section of the A* program whenever a subtree must be visited. This fine control originally came at some cost, insofar as a large percentage of many programs consisted of calls to `traverse`, as illustrated in Fig. 11.

The notable pattern in Fig. 11 is that subtree traversals are interspersed with other activity. This pattern became so common that we changed the semantics of the BNF construct: a grammar rule would continue to have pattern-expression properties, but would also imply the traversal of each subtree. As in `yacc`, semantic actions would be permitted at any point on the right-hand side;[1] traversals would be interspersed appropriately. Fig. 12 shows another way to express the actions of Fig. 11.

1. Since the tree is simply being matched, interspersing semantic actions with grammar symbols cannot introduce ambiguities, as is possible in a `yacc` specification.

```
%{
 ... // some language constructs
_expression: _expression '+' _expression
  {
    printf "Add("
    traverse($$kids[_expression#1])
    printf ","
    traverse($$kids[_expression#2])
    printf ")"
  }
 ... // all other constructs
%% // now, the traverse section
{ dispatch( ) }
%}
```

Fig. 11. Interleaving traversal and output.

```
%{
 ... // some language constructs
_expression:
  {printf "Add(" }
  _expression { printf "," } '+' _expression
  {printf ")" }
 ... // all other constructs
%% // now, the traverse section
{dispatch( ) }
%}
```

Fig. 12. Output together with implicit traversal.

The resulting code resembles yacc with Awk as its statement language rather than C. With this change, the grammar of the subject language serves as a template program. Annotating the template with semantic actions is often the easiest way to build a tool.

IX. Experience With A*

A* has been used in both exploratory research and production software development within Bell Laboratories. Since the ultimate users of our efforts are software developers involved in producing central-office telephone switches for use throughout the world, many of the languages to which A* has been applied are related to telecommunications. To date, the most extensive use has been with versions 4 and 5 of the PRL language, which we mentioned earlier. Other significant projects have used A* to build tools for C and for another internally developed language, SPEC (an internal language for specifying commands and reports for the 5ESS switch). Less significant instantiations of A* have been for A* itself, Awk, the ISO specification language LOTOS [10], and PADL, an internal language for specifying the appearance and interaction of status displays in a telephone switch.

A. PRL4 and PRL5

When we worked with PRL4 and PRL5, our goal was to design a new language and implement its supporting tools; at the same time, we were experimenting with A*.

One improvement to PRL4 that might be credited to A* was a dramatic reduction in the amount of administrative baggage present in PRL4, the earlier incarnation of the language. For instance, PRL4 had syntactic provisions for a change history and for symbol cross-references. Armed with A*, we were able instead to produce relatively simple tools to generate reports from a new and less redundant version of the language.

But as we used A*, its deficiencies were exposed by the ever-increasing complexity of the major PRL tools, particularly the PRL compiler. In addition to motivating a host of performance improvements, experience with the PRL compiler project led us to extend A* in several ways. First, by offering a handful improvements to the Awk statement language itself. Second, by introducing language constructs specifically to support large, complex programs: multiple traversal passes over a parse tree; "pass barriers," a feature for specifying how multiple files should be processed; and the new printf format specifiers described earlier.

At the same time, the convenience of A* became apparent—a number of useful tools were written to analyze PRL4 with little effort. Some examples are tools for construct counting, complexity analysis, and cost modeling.

B. C

Unlike PRL5, a language we were developing, C [11] is a mature and stable language and cannot be improved at the whim of people implementing C-language tools. From the perspective of a tool builder, C's syntax has two fundamental problems, the first and most onerous of which is its use of the C preprocessor. The preprocessor masks various deficiencies of the base C language such as its lack of module system, inline functions, and constants; it is thus extremely rare to find a non-trivial C program that does not rely on the preprocessor. A second problem is the design of C's type definition syntax. With a typedef declaration, the programmer can make any identifier a type name. In order to parse C programs, the lexical analyzer must distinguish type names from ordinary identifiers. This context-sensitivity makes it impossible to parse fragments of C programs. Although our inability to reverse the effects of the preprocessor ruled out source-to-source translation of C, we were nonetheless able to build some interesting tools with **ac**, the A* instance for C. Several of these tools fall into the general class of static source analyzers:

- A tool to compute McCabe's cyclomatic complexity [12]: 10 lines of **ac** code.
- A tool to report the static call graph of a program: 33 lines.
- A tool to produce function prototypes: 130 lines.
- A prototype tool to compare the structure of different fragments of code in a large corpus, looking for patterns to abstract into functions: 200 lines.

Several other tools were constructed with **ac**. One such tool was a simple line-count profiling package. A single **ac** program served as a filter to add instrumentation to the C program under examination and a filter to present a listing with the profile results embedded in the source.

The most ambitious use of A* for C was a prototype symbolic evaluation package for C. This project strained the limits of the A* language and its interpreter with its wholesale copying, manipulation and freeing of symbolic values, represented as parse tree fragments.

C. SPEC

In the case of the internally developed SPEC language, a single programmer worked with one researcher to develop a set of language-processing tools together with a new language. SPEC is used to define, in a single place, an input command, the corresponding terminal acknowledgment, the corresponding output report, and the associated customer documentation for the command and report. Commands and reports conform to the CCITT MML language. In this perhaps most interesting use of A* to date, the developer was able to separate the language design and parsing from the tools needed to translate the single specification into the input command descriptions, output command descriptions, and user documentation in various forms.

D. LOTOS

In the case of the ISO process specification language LOTOS, a fellow researcher was faced with the need to build certain tools. Over the course of a day, we were able to write a grammar for the language and build both an instance of A* for the language and a small but useful tool to translate basic LOTOS specifications into descriptions of action trees amenable to layout with `dot` [13]. This action-tree visualizer illustrated the ideal use of A*, in that it is something for which Awk itself would have been suitable if only it had more sophisticated parsing capabilities—which is precisely what A* offered.

E. PADL

A* was also used to build tools for an internally-developed language called PADL, the PAge Description Language. PADL is used to specify the layout and some of the behavior of the text-screen interface to the 5ESS switch. PADL's structure is much like a traditional assembly language. An `if-then-else` construct is its only concession to structured programming. One of our colleagues embarked on a project to automatically translate PADL descriptions into X Toolkit widget descriptions. The first version of the translator was done in Awk. A second version was written in A*. While manipulating `if-then-else` blocks is somewhat easier in the new version, the costs of developing a grammar and re-writing the translator seem to outweigh this benefit. A* offers more of an advantage when the subject language has complex syntax.

X. Lessons Learned

A* has been used for some two years to build throw-away programs and significant compilers for several languages. Over this time many of its strengths and weaknesses have been exposed; this section makes some observations about A*'s principal drawbacks.

A. An Unsafe Base Language

One reason we chose Awk as our base language is its reputation as a platform for rapid prototyping. Unfortunately, we found that this goal was not advanced by our choice. Awk variables are implicitly initialized and declared, a feature that appears to offer some convenience for the rapid prototyper. In addition, all values in Awk are strings, eliminating the possibility of type errors. The net effect is that an Awk program that works correctly under carefully tested conditions is easily constructed.

However, implicit initialization means that mistyped variables are discovered only during run-time. And the fact that all variables are strings means that other types must be encoded and decoded by the programmer. Therefore genuine type errors can occur even though they cannot be discovered by the interpreter.

The drawback of these conveniences is that an Awk programmer must mentally typecheck his program instead of relying on a compiler to do so. This seems to work rather poorly, particularly when the program grows, or when it persists for so long that not every stage of its development is clearly recalled, or when two or more programmers are building the prototype together. Therefore, since rapid prototyping implies rapid change, it seems that to support rapid prototyping one would prefer languages with exceptionally careful typechecking rather than those which permit errors that can only be detected at run-time.

B. Primitive Pattern-Matching Facilities

Another problem with A* lies in its primitive pattern-matching facilities. Language processors typically do extensive tree analysis, and the patterns in A*'s pattern-action statements typically involve tree and node descriptions. However, wildcards and pattern variables are missing from A*, and its most sophisticated tree patterns (single BNF grammar rules) are syntactically restricted and therefore of limited use. A sophisticated match compiler and appropriate syntax would be a boon to A*.

C. Attribute Evaluation

A* was not envisioned primarily as an attribute evaluator, but in practice a significant amount of effort is expended manually calculating the values of various inherited and synthesized attributes. To the extent that one should be able to specify attributes and rules for their derivation without concern for how they are calculated, A* falls short of a known ideal; incorporating automatic attribute evaluation into A*'s successor would be a valuable contribution.

D. Tree Surgery

A* is based on the traversal of concrete syntax trees. However, for certain applications, concrete syntax trees are inconvenient. For example, an optimizing compiler would typically be organized as a series of passes over increasingly low-level languages, each of which would be represented by a different tree type. Unfortunately, A* does not support such work safely; an improved A* would be better adapted to this kind of work.

E. Semantic Information

A* is based on the syntax of the subject language and does not make non-syntactic information available automatically.

One can imagine, however, that non-syntactic entities like symbol tables ought to be available where appropriate. In the current version of A*, symbol tables and the like must be built by the A* programmer if they are desired, but this can be tedious, making certain tasks more difficult than they ought to be. Incorporating a general way to build such information implicitly would improve A*.

F. Concrete Parse Trees and BNF

A* uses the same BNF as yacc, which permits one to take straightforward advantage of existing yacc code even when it was written for some other tool. However, this BNF does not have any convenient shorthand for specifying an abstract parse tree, leading to idiomatic transformations from concrete to abstract syntax such as tree flattening. If A* were to support an enhanced BNF it could yield more convenient parse trees automatically. At the same time it would be easier to describe new languages.

XI. Conclusion

The essential advantage of A*, like that of other successful high-level languages, is that it encapsulates and hides much of the complexity inherent in a certain class of tools. Our experience with A* indicates that the combination of a high-level statement language with some language processing constructs and useful implicit processing tremendously simplifies language tool construction. In spite of its several drawbacks, A* is useful if one or more of the following hold: the semantic complexity of a language is low; the desired tool implements little of the semantics; or the tool needs to examine only a proper subset of grammatical constructs. Even when these conditions do not hold, A* can be preferable to a combination of lower-level tools such as yacc and C. Because A* is useful in spite of its drawbacks, it seems likely that by addressing its known flaws an even more potent tool can be derived.

References

[1] S.C. Johnson, "Yacc: Yet another compiler compiler," tech. rep., Bell Telephone Laboratories, 1975.

[2] M.E. Lesk and E. Schmidt, "Lex—a lexical analyzer generator," tech. rep., Bell Telephone Laboratories, 1975.

[3] P. Devanbu, "GENOA—a language and front-end independent source code analyzer generator," *Proc. 14th Int'l Conf. on Software Engineering*, 1992.

[4] J.C. Cleaveland, "Building application generators," *IEEE Software*, vol. 5, no. 4, July 1988.

[5] J.R. Cordy, C.D. Halpern, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," *Proc. Int'l Conf. of Computer Languages*, Miami, Fla., pp. 280–285, Oct. 9-13, 1988.

[6] J.R. Cordy and I.H. Carmichael, "The TXL programming language syntax and informal semantics," tech. rep. 93-355, Queens University, 1993.

[7] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, *The AWK Programming Language*. Reading, Mass.: Addison Wesley, 1986.

[8] D.A. Ladd and J.C. Ramming, "Software research and switch software," *Int'l Conf. on Comm. Technology*, Beijing, China, 1992.

[9] B.N. Desai, D.L. Harris, and R.A. McKee, "A formal language for writing data base integrity constraints," *Int'l Switching Symp.*, 1992.

[10] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, pp. 25-59, 1987.

[11] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*. Englewood Cliffs, N.J.: Prentice Hall, 1978.

[12] T.J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 12, Dec. 1976.

[13] E.R. Gansner, E.E. Koutsofis, S.C. North, and K.P. Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, Mar. 1993.

David A. Ladd received the BS and MS degrees in computer science from the University of Illinois at Urbana-Champaign in 1987 and 1989, respectively. He joined AT&T Bell Laboratories in 1989, where he is currently a member of technical staff in the Software Production Research Department. His current research interests are network services and application-oriented languages and environments.

J. Christopher Ramming received degrees in computer science from Yale College (BA, 1985) and the University of North Carolina at Chapel Hill (MS, 1989). He joined AT&T Bell Laboratories in 1987 and is a member of technical staff in the Innovative Services Research Department. His current interests include application languages and their use in software production.