ALADIN: A Scanner Generator for Incremental Programming Environments *

Bernd Fischer
Technical University of Braunschweig,
Institute for Programming Languages and Information Systems,
Gaußstraße 11, D-3300 Braunschweig, Germany

Carsten Hammer
Siemens AG, Corporate Research and Development, Dept. ZFE IS SOF 22,
Otto-Hahn-Ring 6, D-8000 Munich 83, Germany

Werner Struckmann
Technical University of Braunschweig,
Institute for Programming Languages and Information Systems,
Gaußstraße 11, D-3300 Braunschweig, Germany

Mach 19, 1991

Abstract

Summary: A large number of scanner generators have been developed. Since they are restricted to the longest match rule they are unsuitable for an incremental environment. We present the ALADIN-system, which is able to deliver more than a single token if required. Thus, an ambiguity may be passed to the calling instance. Beyond this 'incremental feature', ALADIN is a well-structured and easy-to-understand language. In contrast to existing systems, the desired behavior of the generated scanners is completely specified explicitly. Thus, the specifications are more abstracted than in other systems. A prototype implementation has shown that ALADIN-generated scanners have about the same performance as those generated by Lex.

Key words: lexical analysis, scanner generator, incremental compiler

*Accpeted for publication in *Software – Practice & Experience*

INTRODUCTION

Since the introduction of scanner generators as a general tool for compiler construction [1] in the mid-60's, a vast number of systems have been designed and implemented, for example Lex [2], Flex [3], Rex [4], GRAMOL [5], LEXXO [6] or Alex [7]. This paper describes another system called ALADIN (Advanced Lexical Analyzers DescriptIoN method) which is especially designed for applications in incremental systems.

We will first discuss requirements, which result from the incremental environment and the structure of programming languages, followed by a brief analysis of some typical existing generators. Then, we will describe the basic ideas of the ALADIN system and outline an implementation. Performance data for some languages will be given. A complete specification of the lexical part of Modula-2 [8] will be given at the end of this article.

Unlike other systems which aim at high speed [4] or ease of use [6], ALADIN is designed to be used with a wide range of programming languages. Its original application was the PSG (programming system generator) [10], [11] which is a tool for generating language specific programming environments.*

These programming environments include an incremental compiler and a hybrid-editor which supports a text-oriented and a syntax-directed mode simultaneously. Other components of PSG like library tools or pretty printers may be added. PSG consists of a set of generators (generator system), one for each component, and a set of language-independent table drivers (nucleus), also one for each component. Nucleus and generated tables together form the programming environment which interacts with the programmer.

The table drivers for each component must be completely language-independent to allow switching between different languages during a session. Therefore, the generator must generate pure table data. This prohibits a language definition language (LDL) with interspersed program fragments as used by Lex or Rex.

PROBLEMS IN CURRENT SYSTEMS

Requirements due to an incremental compiler

As already stated, ALADIN has been designed to work in an incremental environment. All components which need to support incremental compilation must meet some additional requirements compared with standard components. For example

*PSG is a joint project of the Technical University of Darmstadt and Siemens, ALADIN is a joint work of Siemens and the Technical University of Braunschweig.

they must cope with problems arising from missing context. For a scanner generator these requirements can be stated as follows.

- A set of possible solutions must be returned, because the context which might be used to solve lexical conflicts might not be known yet. The data type of the result must thus be `SET OF token` instead of `token`.
- A lookahead-operator which includes arbitrary context makes no sense for the same reason and therefore must be discarded entirely. Nevertheless, the next input character could be considered (if available).
- The LDL has to deal with the complete character set including all control characters. The control characters may be legal letters or symbols in some languages (e.g. line wrap in C strings) or may be used to separate incomplete program parts (fragments) from each other.

This will be illustrated by an example. Consider the following Modula-2 fragment

```
◁12. ▷
```

The characters ◁ and ▷ denote the beginning and end of the fragment respectively. A scanner which works according to the usual longest match rule will consider the characters `12.` as a real literal. In an incremental system this behavior (i.e. longest match) will lead to trouble. Subsequent edit-actions may result in a context which confirms the original decomposition (e.g. ◁`12.0`▷) or disproves it. If the fragment is expanded to

```
◁12..15▷
```

tokenizing will have to yield <int, `12`>, <dot_dot, `..`>, <int, `15`>. The scanner has to re-read the characters `12`, causing a loss of efficiency. Still worse, errors may appear. After an expansion to

```
◁TYPE twelvetofifteen = [12. ▷
```

returning the token <real, `12.`> will cause an error although there obviously exists a valid continuation.

Requirements due to the structure of programming languages

Programming languages exhibit wide variety in their lexical structure. This remains true even if only 'modern' format-free languages are considered. Comments, for example, are delimited in as many variations as there are languages. Ada [13] uses the

'endline-comment' which extends from a starting symbol ('--') to the end of the line. Most other languages use 'parentheses-comments' which may sometimes be nested as in Modula-2. The parentheses may consist of several characters and sometimes different parenthesis styles may be used (e.g. Pascal [14]). Other lexical units, for instance notations for non-decimal integer values, exhibit similar variability. So, two conclusions have to be drawn.

- There is no general rule which applies to all languages, not even longest match.
- There is no small and self-contained set of alternatives which fits 'almost all' modern languages. Hence, searching for such a set is pointless.

Nevertheless, the lexical structure of languages share some basic concepts. These concepts include equivalent characters and patterns, irrelevant characters and patterns, non-regular patterns, conflict solution strategies and different kinds of context dependencies (e.g. fixed or arbitrary lookahead, column notation).

How not to do scanner generators and why not

As mentioned above, a number of scanner generators exist up to now. They will be ranked into three classes according to the degree of freedom which is given to the user (i.e. the language definer).

- *Paradigm-based systems* use an existing or imaginary language as a pattern which depicts and fixes the overall-structure of the languages representable in the system. Only some fine-adjustments can be done by the language definer. Those systems such as LEXXO do not use regular expressions at all.
- *Mid-size systems* usually use several description modes at the same time, one for each token class (e.g. reserved words, literals, comments). The description mode uniquely determines the membership to a particular token class and therefore the interpretation of each token. Regular expressions, for instance, may only be used to describe literals. Additional control information such as case sensitivity can be specified by setting flags. Most of the scanner generators, including Alex and GRAMOL, work this way.
- *General purpose pattern recognition systems* like Lex or Rex use regular expressions for the description of all tokens, regardless of the token's class. Arbitrary complex program fragments for each token serve as semantic actions.

A detailed discussion of the different systems is not within the scope of this paper. We will concentrate on the deficiencies which make these systems inappropriate for our purposes.

The major deficiency is their inability to handle lexical ambiguities. Worse, there is no simple way to extend these systems to allow ambiguity, since all these systems use longest match as their 'golden rule'. They will deliver the token that belongs to the longest possible lexeme. If a lexical conflict occurs because a lexeme matches two different patterns, the token to be delivered is chosen in a fixed manner. Lex, for example, uses the pattern's declaration order. Mid-size systems use the semantics of the token. A reserved word always takes priority over an identifier. Resolving lexical conflicts, however, remains a problem even if more lookahead characters are examined. Furthermore, the returned value cannot be a set of tokens, but must be uniquely determined.

Another deficiency arises from the specification languages. 'Non-standard' items like nested comments which constitute simple non-regular patterns cannot be specified in a consistent way. The same is true for tokens that depend on a certain position in a source line such as in FORTRAN.

Besides this systematic flaw there are some more characteristic deficiencies in each class.

- Paradigm-based systems are restricted to a small range of programming languages and therefore obviously not suitable for our purposes.
- To be able to scan all programming languages, mid-size systems must be capable of describing every possible token class and every kind of control information. Otherwise, each new language to be generated potentially needs a new generator version. This completeness is difficult to ensure.
- Lex uses arbitrary program fragments extensively. It is more a preprocessor tool than a real generator. Moreover, the Lex specification language is difficult to use, even for an experienced user. Consider for example the rule

  ```
  \/\*\/*([^\/\*]\/*|\*+)*\*\/ ;
  ```

  which is appropriate for C's comments and also describes the corresponding action (i.e. ignoring comments).

SOLUTIONS IN ALADIN

Most of the problems in existing systems arise from too careless language design. The definition languages consist merely of a collection of add-on-features which might be useful in special situations but they do not have a uniform style. Adding a new

'incremental feature' makes it even less uniform. Hence, a new approach which uses only a few necessary components was taken.

One of the design objectives of ALADIN was a strict separation of pattern description and control flow elements so that they could be treated independently, with no dependencies on each other. Systems that do not have this separation exhibit the problem that a particular description style strongly influences the pattern's interpretation.

The second objective is that the pattern description style should be uniform throughout the entire specification. The only description element in ALADIN is the regular definition which is used regardless of whether a character set, a keyword or a literal is defined. A regular definition consists of a defined name and a defining regular expression which is associated with the name. If a name is used in a defining expression it will be replaced by its own associated expression.

Even non-regular patterns such as nested comments may be specified in the same way. The defined name itself can be used in its own defining expression like in

```
com -> "{" (letter | com)* "}";
```

Hence, we get a consistent improvement of regular expressions and do not need any '`NESTED`'-commands as in GRAMOL or explicit counters as in Lex. All necessary actions are done internally, by the generator itself. However, some restrictions must be imposed for efficiency. These will be described later.

The final design objective is that there is no implicit control flow procedure. ALADIN allows the language definer to manipulate the behavior of the generated scanners in her/his own way. Only two mechanisms are required for this: grouping and attributing.

Grouping means that tokens which share the same lexical characteristics (e.g. priority levels) are 'pooled' together. All characteristics of a group must be given explicitly in the form of recognition constraints. Arbitrary combinations of constraints are possible as long as there are no contradictions. This grouping mechanism seems to be similar to standard token classification (literals, delimiters, etc.) used by most existing systems but it is quite different for two reasons. First, an arbitrary number of groups with arbitrary constraints can be specified, thus allowing the user to model complex structures. Second, the constraints do not result from any kind of implicit interpretation like the order of the groups. This increases not only the flexibility but also the clearness and reliability of the specifications.

Attributing is a mechanism for specifying a particular policy, either for a single token or for a group. Token level attributes are activated when a pattern matches. The attributes pass information to the generator system (e.g. 'deliver another token if this pattern matches') or cause some action within the driver (e.g. 'ignore this token

if it is matched'). Attributing on the group level is a suitable way for the specification of recognition constraints for a group.

We call the way ALADIN tokenizes the input *multiple match*. Every recognized token is included in the set of returned tokens unless one of the following two situations appears.

- A particular policy has been explicitly specified.
- Tokens defined in the same group share the same lexical characteristics and may be seen as different lexemes of a 'supertoken'. Thus, within a group the longest match applies and a single token per group is returned.

Tokenizing continues until the next input character does not fit any of the patterns.

Due to the multiple match mechanism and the ability to specify an arbitrary number of groups, no special 'incremental feature' is required. Tokens which might cause problems because of missing context need only be specified in different groups. Thus, the lexical conflicts are passed to the calling instance which is able to resolve the conflicts.

The following section briefly explains the ALADIN syntax. For the complete syntax see reference [12].

STRUCTURE OF THE LANGUAGE

Lexical elements of ALADIN

The lexical structure of ALADIN is very compact, nevertheless it remains clear. It is a format free language whose tokens are separated by at least one white space character. The white spaces have no other meaning, particularly they are not delimiters for the patterns as in Lex. Any white space may be replaced by an Ada-like comment. The identifier pattern follows the usual standard — a letter followed by arbitrarily many letters, digits and underscores. Only lower case letters may be used for identifiers whereas ALADIN keywords are composed of upper case letters.

ALADIN uses two kinds of literals, strings and cardinals. A string is an arbitrary sequence of characters enclosed in double quotes. There is no character literal. Instead, strings of length one are used.

Some special characters are used as operators, either as single character operators or as compound operators such as `+`, `*`, `->`, etc.

Pattern description

As mentioned above, the only description element in ALADIN is the regular definition, for example

```
name -> regular? expression*;
```

Regular definitions that are members of groups are treated like token definitions. Other definitions are auxiliary and only facilitate the specification. The syntax of the regular expressions roughly follows the Lex model, i.e. postfix operators are used. Some things have been changed to improve legibility.

A regular expression extends from the arrow ('->') to a semicolon. Arbitrary spaces, tabs and newlines may be used to 'style up' complex patterns. All terminal symbols must be strings. Thus, 'definition expansion' as it is called in Lex needs no longer to be tagged. Each name used in a pattern description is replaced by its definition. Forward references are allowed. ALADIN also uses predefined identifiers, similar to `true` and `false` in Pascal.

Character sets are used for modeling equivalence of characters. Members of a character set are either single characters or identifiers which are also defined as single characters or character sets themselves. Thus, no explicit set union operator is necessary. Only the '\'-operator for set difference is required. In connection with the predefined set '`all`' which contains all representable characters it is used for set complementation, similar to the mechanism used in Alex.

```
-- character set examples
oct_digit -> {"0", "1", "2", "3", "4", "5", "6", "7"};
digit     -> {oct_digit, "8", "9"};
non_digit -> all \ digit;
```

If a name is used in its own pattern description the standard expansion method does not work and another mechanism has to be defined. Two different situations are possible. If the pattern describes a regular language, it must be a left or right recursive expression. This is internally transformed into an equivalent regular expression with postfix operators. Thus, the left recursive definition for C's octal integers

```
oct_int -> oct_int oct_digit
         | zero;
```

will internally be transformed into its equivalent

```
oct_int -> zero oct_digit*;
```

Yet another situation arises if such transformations are impossible because the 'regular expression' describes a formal language which is in fact not regular. For efficiency only restricted patterns can be specified. On the lexical level only nested expressions should be evaluated. All other work is done on the parser level. Legal patterns must not lead to any conflicts between normal and recursive processing mode. This is guaranteed if the following two constraints are met. First, each alternative of such a pattern may contain at most one recursively defined identifier. Second, for each such identifier a non-recursive alternative must be given. The first-set of this alternative and those of the expressions preceding and following the use-occurrence of the identifier have to be mutually disjoint. For example,

```
wrong -> "{" wrong* "}"
       | "{";
```

meets only the first constraint but not the second one and is thus not a legal pattern whereas nested comments as in Modula-2 (see appendix) are legal patterns.

Control flow

Control flow elements in ALADIN specify how the defined patterns shall be treated and thus control the behavior of the generated scanners. As mentioned above, only two basic control flow concepts are required. They are independent from each other and independent from the pattern description.

A group of tokens is formed by 'GROUP' and 'ENDGROUP', which surround and name an arbitrary number of regular definitions, e.g.

```
GROUP res_words;
  and  -> "AND";
  ...
  with -> "WITH";
ENDGROUP;
```

Recognition constraints for the group must be specified following the group's name, separated by commas. ALADIN uses three types of constraints: the priority, the prefix and the context constraint.

The *priority constraint* is used to solve lexical conflicts. Every token has a priority level which results from the language definition. It determines which token is returned if a lexical conflict occurs. If reserved words for example have a higher priority level than identifiers, the group definition

```
GROUP identifiers,
```

```
  PREFERRED BY res_words; -- priority constraint
  id -> letter {letter, digit}*;
ENDGROUP;
```

will model this behavior. Otherwise, due to the multiple match mechanism, not all conflicts have to be resolved and the specification of a total priority order is not necessary. In the case of a conflict all tokens are returned which are not preferred by any other token involved in this conflict.

Normally, multiple match implies that a token is also recognized if it occurs as a prefix of another token which is member of another group. Sometimes this behavior is desired (remember for instance the Modula-2 example earlier or the famous FORTRAN-DO5I-example [9]), but sometimes it is not. If prefix recognition is not desired, it can be suppressed by the *prefix constraint*, e.g.

```
GROUP res_words,
  FOLLOWING all \ {letter, digit}; -- prefix constraint
  and  -> "AND";
  ...
  with -> "WITH";
ENDGROUP;
```

The meaning of a prefix constraint is that a token of such a constrained group will only be recognized if the character next to the respective lexeme is a member of the specified character set. This is a kind of lookahead but due to the restriction of sets it is restricted to a single character. This character is either a normal source text character or it signals the end of the fragment. Thus, it may be considered in each case.

The *context constraint* handles complex context conditions. These are conditions that could not be checked by merely considering the next input character as for example FORTRAN's column sensitivity. A special checking routine which is part of the driver is required for each complex context condition. A condition which yields false delimits the lexeme, regardless whether the next real input character matches or not. The only complex context condition which is currently supported by ALADIN is the column-dependent notation as it is used in FORTRAN, e.g.

```
GROUP fortran_key_words,
  START #7,  -- context constraints
  STOP  #72;
  ...
ENDGROUP;
```

Token level attributes

Besides the group level attributes, ALADIN knows about token attributes. They are appended to the respective defining expression, also with a comma. The most important token level attributes are the result, ignore and perform attribute.

Normally, the pattern name is also the name of the token. If several alternative patterns fit the token, the alternative operator should be used. Sometimes the alternatives require very different policies. Remember the Modula-2 example of 'AND' and '&' which yield both the same token 'and'. Nevertheless, it is not possible to specify a pattern

```
and -> "AND" | "&";
```

because '◁&a▷' first yields <and, &> and subsequently <id, a>, whereas '◁ANDa▷' yields only <id, ANDa>. The *result attribute* changes the returned token name to the value given in the result attribute as in the following example.

```
short_and -> "&", RESULT and;
```

Another token attribute, the *ignore attribute* is required for such unpleasant language constructs (at least from the lexical point of view) as comments or white spaces. They are not really tokens because they should not be returned as a token by the scanner, but this depends on the language (consider for instance OCCAM's indentation token) or even on the environment in which the scanner has to work, e.g. comment management. Thus, the ignore attribute which discards the just recognized token and restarts the scanner can be used to manage this problem, as the following example for ALADIN comments shows.

```
comment -> "--" (all \ eol)* eol, IGNORE;
```

Every scanner generator is faced with the problem of what to do with patterns which cannot be described by the normal pattern description method, e.g. FORTRAN's Hollerith strings. A system which claims to be general cannot simply ignore these exceptional patterns. The usual solution is to support special user-written routines, but like Lex fragments this conflicts with the goal of a language-independent driver. Moreover, the correctness of the routines depends on the user. We decided to offer a library of handler routines to the language definer which cover the most common problems. These routines are a fixed part of the table driver. They can be called by the *perform attribute*, e.g.

```
comment -> "--", PERFORM read_end_of_line;
```

which has the same effect as the example above. Using library routines also makes the generated scanners more efficient.

In addition to tokenizing, attribute evaluation is another major task for a scanner. The scanner has to return not only the symbol or token code but also several attributes. Almost all scanners return the lexeme (i.e. the matching source text) and so does ALADIN.

Two other attributes that are common in hand-written scanners are not appropriate for generators. The insertion of identifiers in a symbol table requires a lot of context information and must be done by the parser. The calculation of an internal bit-representation depends on the hardware and should not be done by the scanner. Only 'calculations' on a mere text-transformation-level are supported by ALADIN. This normalized lexeme form cannot only be used for pretty-printing purposes but also for semantic analysis. The lexeme is transformed letter by letter, according to user-specified substitution rules.

Other attributes for special purposes may be added if they are needed, e.g. for the management of a lexeme pool. In contrast to other language definition languages, this may be done without any changes to the basic language concept.

IMPLEMENTATION ASPECTS

A complete ALADIN system consists of a generator part and a driver or table interpreter part. A prototype was written in Pascal-XT, a Pascal superset, and runs under SINIX, the Siemens version of UNIX. Both parts together consist of approximately 8000 lines of code. The implementation took about four months.

ALADIN's front end uses standard compiler techniques and could be replaced by any generated front end. The scanner of the front end of course is generated by ALADIN itself.

The back end of the ALADIN generator generates a non-deterministic finite automaton (NFA) which is subsequently made deterministic (DFA). It primarily uses the algorithms described by Aho *et al.* [9] with some changes for efficiency.

The generation follows the Thompson algorithm but we use a slightly different representation of the transitions. The original algorithm asserts that each state has exactly one terminal character transition or at most two epsilon transitions, thus allowing a memory-saving implementation of the 'transition lists' in arrays. On the other hand this will cause character set transitions to fan out into different states and thereby drastically increase the number of generated states. Thus we decided to implement 'real' transition lists. A second difference is due to the extension to non-regular patterns. A straightforward implementation would count the number of

opening and closing parentheses. But it also has to determine which counter must be updated. Our automatic implementation uses a stack instead of several counters. We will describe it using the simplified example of nested comments, specified by the definition

```
com -> "{" (letter | com)* "}";
```

First an automaton without recursively defined identifiers (in the above example `"{" letter* "}"`) is built using the normal routines. The final states of each subautomaton ($z_1 - z_3$, see Fig. 1) which are uniquely determined due to the Thompson construction will be required subsequently. The automaton for a recursively defined identifier will be constructed as follows.

- If its opening parenthesis is detected in the input stream, the automaton for this identifier will be called like a subprogram. For this, the 'returning address' (i.e. the state from which to continue) must be pushed on a stack. Thus, an automaton for the opening parenthesis (here "{") is built. Its final state (z_4) is marked with the action 'push the continuation state (z_1) on the stack'.

- The 'automaton call' is done by an ϵ-transition from z_4 to z_1, because z_1 is the final state of the opening parenthesis subautomaton of the recursively defined identifier.

- Returning from the 'automaton call' means popping the continuation state off the stack. Thus, the final state of the closing parenthesis of `com`, which is z_2, is marked with the action 'try to pop the continuation state (z_1)'. If it can be popped, the automaton enters the continuation state (z_1) via an ϵ-transition.

- If the stack is empty, no 'automaton call' has to be finished and the pattern is complete. Thus, the token may be accepted.

This scheme could easily be applied to any allowed case. It is optimal in a sense that only the minimum amount of administration must be done. Only states z_2 and z_4 require additional actions. The largest part of the input is processed as usual.

In a last step the resulting NFA is made deterministic by means of the subset construction. Due to the structure of the generated NFA some modifications have been made in order to save execution time. The subset algorithm has two inner loops, an explicit loop which loops over all characters *ch* and an implicit loop which loops over all transition lists and non-deterministic states in order to calculate *move(state, ch)*, the latter loop being the innermost. The average length of the transition lists is for practical cases 5.5 entries. Hence, for most characters running through the

Language	ND-states (essential)	D-states	Time required
ALADIN	206 (136)	120	45 sec.
Algol68	626 (358)	294	13 min.
C	637 (329)	268	12 min.
Fortran	5182 (1235)	623	20 min.
Modula–2	469 (290)	254	11 min.
Pascal	622 (247)	212	13 min.
PL/I	1694 (1188)	891	121 min.

Table 1: Generator performance

lists is in vain. We switched the order of these loops so that every transition need be considered only once. The price for this time-saving is a higher memory demand. The transitions of a given deterministic state cannot be calculated until any transition of the respective non-deterministic states has been considered. Thus, all states which result from the calculation of *move(state, ch)* must be stored intermediately.

Each state, even non-deterministic, may have some marks, e.g. accepting state and token code. A deterministic state generally inherits all marks of the non-deterministic states it is composed of. Nevertheless, if several accepting states are merged together, the priority constraints must be regarded. Another difficulty are context constraints. If states with different context constraints (e.g. overlapping columns) were grouped together, the evaluation of the context constraints might yield contradictory results. Thus, only groups with equivalent constraints may be considered at the same time. For most cases (i.e. format free languages) this will lead to a single, coherent DFA, but in some cases several distinct DFAs may appear which must be interpreted simultaneously.

No state minimization or table compression takes place. These steps may be added in later versions of the ALADIN system.

The subset construction has the well-known exponential time complexity but this affects — also well known — only some abstruse patterns. In practical cases this algorithm is fast enough. Table 1 shows timings and automata sizes for some programming languages. More than 90% of the whole generation time is spent on the subset construction. All times were measured on a Siemens-MX500 computer.

The performance of the generated scanners which is commonly of more interest than the generation time of the scanner was measured for two languages (ALADIN and Pascal) and a series of different source files, ranging from small (5 KBytes) ALADIN specifications to extensive Pascal programs up to 700 KBytes. These per-

formances were compared with those of two other scanners, a straightforward hand-coded scanner for the first version of ALADIN and a Pascal scanner, generated by Lex. The former was implemented within a few hours without any optimization and processes about 1700 cps (characters per second), the latter 3500 – 4500 cps, depending on the source file size. Different versions of table drivers have been tested. The fastest version scans about 3300 cps Pascal sources, but for smaller sources this drops to 2600 cps. Effectively, 70–80% of the respective Lex performance is achieved. ALADIN sources are processed faster, but due to their smaller size the highest speed was 3000 cps. This difference is caused by the more careful lexical design of ALADIN thus reducing the rate of characters processed multiple times from 35% in Pascal to 15%. Other versions with different additional features have been tried. Table compression was simulated by calculating the next state twice, but the performance loss was surprisingly low. The performance dropped only by about 5%. In most systems, such as Lex, there is no correspondence between tokens and source text positions. Maintaining this correspondence is optional in ALADIN and causes performance loss of about 15%.

The generated scanners are not as fast as those generated by special high speed generators such as Rex. But the performance is high enough, especially in an incremental system where the sources to be tokenized are not very big. Further speeding-up could be achieved. Results by Grosch [4] show that an implementation in C is more than 60% faster than an equivalent Modula-2 implementation. Similar results seem to be possible in our case, too.

CONCLUSIONS

We have demonstrated that a scanner that has to work within an incremental environment has to fulfill some additional requirements. These requirements, which include controlled behavior in the absence of necessary context influence the corresponding scanner generator. We presented the ALADIN-system which avoids this problem by the multiple match rule. An ALADIN generated scanner is able to deliver more than a single token if required or if a unique determination is impossible due to the lack of context. This behavior can be controlled by the scanner specification. Even a specification according to the traditional longest match rule is possible.

Beyond this special purpose we believe that ALADIN as a lexical analyzer description method has several advantages over existing systems. The language definition language itself is compact and self-contained and thus easy to understand. It is based on only two basic principles, grouping and attributing which might be combined arbitrarily. The behavior of the generated scanner is explicitly specified and does not

result from any abstruse interpretation of the specification, not even from the order of the definitions. The pattern description is specified uniformly using regular expressions with a consistent extension for some non-regular patterns. Finally, the scanners can be generated fully and do not need any manually implemented semantic actions. Hence, ALADIN specifications are more abstract than specifications written in other languages.

Acknowledgments

The authors would like to thank the referees for their valuable comments on an earlier version of this paper.

References

[1] W.L. Johnson *et al.*, 'Automatic Generation of Efficient Lexical Processors Using Finite State Techniques', *Comm. of the ACM*, **12**, 805–813, (1968).

[2] M. E. Lesk, *Lex – A Lexical Analyzer Generator*, Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[3] V. Paxson, *Flex – Manual Pages*, Public Domain Software, 1988.

[4] J. Grosch, 'Efficient Generation of Lexical Analyzers', *Software — Practice and Experience*, **19**, 1089–1103, (1989).

[5] C. Genillard and A. Strohmeier, 'GRAMOL – A Grammar Description Language for Lexical and Syntactical Parsers', *SIGPLAN Notices*, **23**, 103–122, (1988).

[6] P. Schnoorf, 'Dynamic Instantiation and Configuration of Functionally Extended, Efficient Lexical Analyzers', *SIGPLAN Notices*, **23**, 93–102, (1988).

[7] H. Mössenböck, 'Alex – A Simple and Efficient Scanner Generator', *SIGPLAN Notices*, **21**, 139–148, (1986).

[8] N. Wirth, *Programming in Modula-2*, 3rd, corrected edition, Springer, Berlin, 1985.

[9] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, Reading, MA., 1986.

[10] R. Bahlke and G. Snelting, 'The PSG System: From Formal Language Definitions to Interactive Programming Environments', *ACM Trans. on Programming Languages and Systems*, **4**, 547–576, (1986).

[11] *Language Definer's Guide to PSG*, Report PI-R6/88, Technische Hochschule Darmstadt, Darmstadt, 1988.

[12] B. Fischer, *Design und Implementierung eines Scanner-Generators im Rahmen des PSGII-Projektes*, Diplomarbeit, Technische Universität Braunschweig, Braunschweig, 1990.

[13] G. Goos and J. Hartmanis *ed.*, The Programming Language Ada Reference Manual, ANSI/MIL-STD-1815A-1983, *Lecture Notes in Computer Science*, **155**, Springer, Berlin, 1983.

[14] K. Jensen and N. Wirth, *Pascal User Manual and Report*, 3rd edition, Springer, New York, 1985.

APPENDIX A: ALADIN-SPECIFICATION OF MODULA-2

```
LEXIS;   -- Modula-2

-- character set definitions
letter    -> {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j",
              "k", "l", "m", "n", "o", "p", "q", "r", "s", "t",
              "u", "v", "w", "x", "y", "z",
              "A", "B", "C", "D", "E", "F", "G", "H", "I", "J",
              "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T",
              "U", "V", "W", "X", "Y", "Z"};
oct_digit -> {"0", "1", "2", "3", "4", "5", "6", "7"};
digit     -> {oct_digit, "8", "9"};
hex_digit -> {digit, "A", "B", "C", "D", "E", "F"};
ws_char   -> {bol, eol, tab, " "}; -- white spaces
asterisk  -> "*";
s_quote   -> "'";

-- auxiliary definitions
inner_com  -> all \ {asterisk, right_par} right_par* | asterisk+;
signed_int -> {plus, minus}? digit+;

GROUP white_spaces;
  comment -> "(*" right_par* (inner_com | comment)* "*)", IGNORE;
  ws      -> ws_char+, IGNORE;
```

```
ENDGROUP; -- white_spaces;

GROUP reserved_words,
  FOLLOWING all \ {letter, digit};
  and  -> "AND";
  ...
  with -> "WITH";
ENDGROUP; -- reserved_words

GROUP delimiters;
  equal      -> "=";
  not_eq     -> "#" | "<>";
  greater    -> ">";
  greater_eq -> ">=";
  less       -> "<";
  less_eq    -> "<=";
  short_and -> "&", RESULT and;
  short_not -> "~", RESULT not;
  plus   -> "+";
  minus  -> "-";
  times  -> "*";
  divide -> "/";
  assign_op -> ":=";
  deref_op  -> "^";
  left_par  -> "(";
  right_par -> ")";
  left_sqb  -> "[";
  right_sqb -> "]";
  left_bra  -> "{";
  right_bra -> "}";
  dot       -> ".";
  comma     -> ",";
  semicolon -> ";";
  colon     -> ":";
  dot_dot   -> "..";
  bar       -> "|";
ENDGROUP; -- delimiters

GROUP identifier,
```

```
  PREFERRED BY reserved_words;
  id -> letter {letter, digit}*;
ENDGROUP; -- identifier

GROUP integer_literals;
  integer  -> digit+;
  oct_int  -> oct_digit+ "B";
  oct_char -> oct_digit+ "C";
  hex_int  -> digit hex_digit* "H";
ENDGROUP; -- integer_literals

GROUP other_literals;
  real     -> digit+ dot digit* ("E" signed_int)?;
  string   -> quotes (all \ {quotes, eol})* quotes
            | s_quote (all \ {s_quote, eol})* s_quote;
ENDGROUP; -- other_literals

ENDLEXIS; -- Modula-2
```