Dynamic Parsers and Evolving Grammars

S.Cabasino, P.S.Paolucci, G.M.Todesco.

INFN, Sez. di Roma, Gruppo Ape, Dip. di Fisica, Universita' di Roma “La Sapienza”, P.le Aldo Moro 5, 00185 Roma, Italy.

Abstract

We define “evolving grammars” as successions of static grammars and dynamic parsers as parsers able to follow the evolution of a grammar during the source program parsing. A growing context-free grammar will progressively incorporate production rules specific for the source program under parsing and will evolve following the context created by the source program itself toward a program specific context-free grammar. Dynamic parsers and growing grammars allow a syntactic-only parsing of programs written in powerful and problem adaptable programming languages. Moreover dynamic parsers easily perform purely syntactic strong type checking and operator overloading. The language used to specify grammar evolution and residual semantic actions can be the evolving language itself. The user can introduce new syntactic operators using a bootstrap procedure supported by the previously defined syntax.
A dynamic parser ("Z $_Z$Parser") has been developed by us and has been successfully employed by the APE 100 INFN group to develop a programming language (“ApeseLanguage”) and other system software tools for the 100 GigaFlops SIMD parallel machine under development.

Introduction

This paper reports a theoretical starting framework and some practical results about “evolving grammars”, i.e., grammars that evolve at parse time. Our approach has several points of contact with the work of Boris Burshteyn on modifiable grammars ([G1],[G3])
and with that of Henning Christiansen on adaptable grammars ([G2]), and has been independently developed by us (1989-1992) during the design and implementation of a compilation system based on grammar evolution at parse time ([Z1],[Z2],[Z3]).

We anticipate that using our evolving grammar approach, the language used to specify grammar evolution and residual semantic actions can be the evolved language itself. The user can then introduce new syntactic operators and perform semantic actions using a bootstrap procedure supported by the previously defined syntax.

Our interest in formal language theory development is mainly connected with the definition of innovative computer programming languages and the implementation of suitable compilers, in the framework of the INFN APE100 project ([A1][A2][A3]). The target of the APE100 project is the development of a 100 Gigaflops, 2048 nodes, SIMD parallel computer dedicated to physics numerical simulations. A 6 Gigaflop, 128 nodes APE100 machine has been running numerical simulation from the beginning of the 1992.

In the mid 1950s Noam Chomsky gave a mathematical model of certain classes of grammars in connection with his study of natural languages [L1].Starting from the 1960s the "formal language theory" has extended its applications to several areas in mathematics and computer science. In our view the conceptual framework of evolving grammars (i.e., grammars changing in time, as opposed to conventional static grammars) follows as a natural extension from the Chomsky grammar definition. Evolving grammars in turn generate evolving languages and dynamic parsers, i.e., parsers able to follow grammar evolution during the parsing of a source program.

We found this dynamic approach useful to solve a class of typical problems in programming language definition and compiler design. Moreover the applications written by the APE100 users using an evolving language are several times shorter than programs written in conventional programming languages, due to the easy introduction of new operators, statements and data types. Furthermore an evolving language always guarantees syntactic type checking on operands. The opinion of the APE100 user community is that these factors greatly enhance the quality of the resulting application programs.

Let us also spend some words about the classes of typical problem in compiler design that can be faced using a dynamic parsing approach.

Compilers [C1][C2] are programs whose task is to translate a program written in a source language into some other destination language (e.g. machine dependent assembler language). To properly execute this translation a precise definition of the language to be recognised (i.e., the syntactical structure of the source language) and the semantics of the programming language (that is the meaning of each construct and the way to translate it in terms of the destination language) must be specified.

One problem of source language syntax specification and recognition is that some statements in a source program may have the power to create a specific context that affects the syntactical and semantical acceptability of the following statements in the source program. In some sense the "meaning" of one statement creates a context in which the other statements of the program must be meaningful and syntactically acceptable. Examples of statements that create a context are declarations of identifiers (that associate the declared identifier with a specific data type) and the description of argument list for procedures. These two examples raise some problems: checking that identifiers are declared before their use in a program and used only in operations that are legal for that data type; checking that the number and type of formal parameters in the declaration of a procedure agree with the number and type of actual parameters in a call to the procedure.

One of the fundamental grammar classifications distinguishes context-free grammars from context-sensitive grammars. The syntax of common programming languages (like FORTRAN, PASCAL or ADA) cannot be completely described by static context-free grammars as follows from the previous considerations.

Due to the difficulty of parsing static context-sensitive grammars, the definition of the syntax of a programming language has been usually given in two parts. The first one is a truly syntactic definition, the second one is an informal mixing of syntax and semantic.

In practice programming language syntaxes have commonly been described by context-free grammars. Therefore it has been customary to place some context-sensitive restriction on the context-free grammar. Such restrictions include data type specific identifier lists, type-matching rules for identifiers and the requirement that a call of a procedure contains exactly as many arguments as there are parameters in the definition of the same procedure.

Compilers often check the context-sensitive part of statement syntaxes during translation phase, that is during semantic processing.

Because semantics and context-sensitive syntax have been so closely associated in both the description and the translation of a language, it has become customary to apply to both the term "semantics".

Evolving grammars offer a different approach to the semantic challenge.

We don't force a predefined static grammar to recognize all the possible contexts created by individual programs. Rather we introduce growing grammars that will adapt themselves to the specific context.

The growing context-free grammar will progressively incorporate production rules specific for the source program under parsing and will evolve following the context created by the source program itself toward a program specific context-free grammar.

The definition of dynamic parsers allows a truly syntactic-only parsing of common programming language programs. Moreover dynamic parsers easily perform syntactic only operator overloading and strong type checking.

By means of dynamic parsers, it is possible to define "evolving" grammars and languages with the aim to move the context-sensitive part of the grammar from the semantic treatment toward a syntactic one.

A dynamic bottom-up translation parser ("Z_ZParser") has been developed by us and has been successfully employed by the APE 100 INFN group to develop an evolving programming language ("Apese Language") and other system software tools for the 100 GigaFlops SIMD parallel machine under development.

User programs written in the evolving language Apese (simulating fluidodynamic systems, neural networks, subatomic particles) are now running on APE100 parallel super computers.

We plan to describe how to design a real life dynamic parser and compiler in a forthcoming document while the incremental language driving the Z_ZParser growth is described in the document "Z_ZLanguage"[Z2]. The first end-user programming language designed using the Z_Z Parser and Z_Z Language is covered by the document "Apese language"[Z1].

Evolving Grammars

A conventional (<u>static</u>) grammar G is usually defined as a 4-tuple $G=(V_n, V_t, \Phi, S)$ where:

- V_t is a finite non empty set of symbols called the terminal alphabet; the symbols in V_t are called terminal symbols;
- V_n is a finite non empty set of symbols called non-terminals (they are used in Φ to describe the syntactic structure);
- Φ is a finite non-empty set of "production rules", i.e., relations $\alpha \rightarrow \beta$ where:
  $\alpha \in (V_t \cup V_n)^* V_n (V_t \cup V_n)^*$ and $\beta \in (V_t \cup V_n)^*$
- S is a distinguished element of V_n called starting symbol.

A **grammar evolution E** could intuitively be conceived as a succession of static grammars:

$$E = \{G^i = (Vn^i, Vt, \Phi^i, S)\ ,\ i=0..n\}.$$

In this work we will assume that Vt is fixed, and that Vn and Φ evolve by successive accretion of new elements, i.e., $\Phi^{k+1} \supset \Phi^k$, $V_n^{k+1} \supset V_n^k$, so that we can describe the evolution as a succession of steps:

$$G^{k+1} = G^k + \Delta G^k \text{ where } \Delta G = (\Delta V_n, \Delta\Phi)$$

and therefore

$$G^{k+1} = (V_n^k \cup \Delta V_n^k, Vt, \Phi^k \cup \Delta\Phi^k, S).$$

We could now introduce classes of grammatical evolutions, drawing our inspiration from the Chomsky grammar classification scheme. The first class could be the "unrestricted evolving grammar class", containing all generic evolving grammars, without any restriction on the type of the production rules. Another class is the context-free grammatical evolution class.

Note that if G^k is a context-free grammar and $\Delta\Phi^k$ is a set of context-free production rules, the $G^{k+1} = G^k + \Delta G^k$ resulting grammar will be also a context-free grammar. From this point on our discussion will be restricted to context-free grammatical evolutions. We will write $\psi \underset{G^i}{\Longrightarrow} \sigma$, $\psi \overset{*}{\underset{G^i}{\Longrightarrow}} \sigma$, $\psi \overset{R^*}{\underset{G^i}{\Longrightarrow}} \sigma$ to specify direct derivations, derivations and rightmost derivations according to G^i $(\psi,\sigma \in (V_t \cup V_n^i)^*)$.

We would like now to specify a mechanism able to generate grammar evolutions. Therefore we will associate to some of the production rules of Φ^i the desired ΔG's writing:

$$\Delta G^i(\Phi_\mu^i) = (\Delta V_n^i(\Phi_\mu^i), \Delta\Phi^i(\Phi_\mu^i)).$$

Informally speaking $\Delta G^i(\Phi_\mu^i)$ specifies the new non-terminals and production rules to be added to the G^i grammar to generate G^{i+1} when the rule Φ_μ^i of G^i is reduced. We will write $\overset{R}{\underset{\Delta G^n}{\Longrightarrow}}$ to specify a rightmost direct derivation in G^n using a single production rule of G^n associated with ΔG^n.

Definition. An **evolving grammar** $\mathbf{Z_a}$ is a grammar G_a^0 having at least one the production rules Φ_μ^0 associated to a non-empty $\Delta G^0(\Phi_\mu^0)$.

Definition. We introduce the definition of **grammatical evolution** starting from G_a^k as a succession of grammars connected by evolution steps:

$$E(G_a^k) = \{G^i \mid G^{i+1} = G^i + \Delta G^i(\Phi_\mu^i),\ i \geq k;\ G^k = G_a^k\}.$$

where $\Delta G^i(\Phi_\mu^i)$ is the grammatical change associated with the production Φ_μ^i of the grammar G^i.

Definition. The **rightmost evolved derivation** $\psi \overset{R^*}{\underset{G^n .. G^k}{\Longrightarrow}} \sigma$ from ψ to σ along a grammatical evolution path from G^k to G^n is defined as $\psi \overset{R^*}{\underset{G^n}{\Longrightarrow}} \alpha \overset{R}{\underset{\Delta G^{n-1}}{\Longrightarrow}} \beta \overset{R^*}{\underset{G^{n-1}}{\Longrightarrow}} \ldots \overset{R}{\underset{\Delta G^k}{\Longrightarrow}} \omega \overset{R^*}{\underset{G^k}{\Longrightarrow}} \sigma$.

Note that grammar grows from right to left (i.e., $G^k \prod G^n$).

Definition. We define the **evolving language** Z_ZLanguage generated by an evolving grammar Z_Z the set of strings x:

$$Z_Z\text{Language} = \{\ x \mid x \in V_t^*;\ S \underset{G^n \cdot\cdot G^o}{\overset{R^*}{\Longrightarrow}} x,\ n \geq 0;\ G^0 = G_Z{}^0\}.$$

Note that an evolving grammar is generally more powerful than a context -free grammar. In the following example we will generate using an evolving grammar a language which is impossible to describe by means of a context -free grammar. The language we want to describe is the set of all xyx where x $\in (a|b)^*$. This language is not context -free and abstracts the problem of checking the declaration of the identifiers before their use in a program. That is the first x in xyx represents the declaration of a n identifier and the second one represents its use. An evolving grammar able to generate this language could be:

$G_a{}^0 = (V_n{}^0, V_t, \Phi^0, S)$ where

$V_n{}^0 = \{S, D, I, U_0\}$

$V_t{}^0 = \{\mathbf{a}, \mathbf{b}, \mathbf{y}\}$

$$
\begin{aligned}
\Phi^0 = \{ & S \rightarrow D\ U_0 \\
& D \rightarrow \mathbf{y}\ \Delta G^i = (\{\}, \{U_i \rightarrow \varepsilon\}) \\
& D \rightarrow I\ D \\
& I \rightarrow \mathbf{a} \qquad \Delta G^i = (\{U_{i+1}\}, \{U_i \rightarrow \mathbf{a}\ U_{i+1}\}) \\
& I \rightarrow \mathbf{b} \qquad \Delta G^i = (\{U_{i+1}\}, \{U_i \rightarrow \mathbf{b}\ U_{i+1}\})\ \}
\end{aligned}
$$

The non-terminal D can generate any sequence of **a**'s and **b**'s terminated by **y**, while the non -terminal U_0 will generate only the same sequence generated by D. During the generation of the string **a b b y a b b** the grammar changes four times:

$G^4 = G_a{}^0 + \Delta G^{0:4}$

$\Delta V^{0:4} = \{U^0, U^1, U^2, U^3\}$

$$
\begin{aligned}
\Delta\Phi^{0:4} = \{ & U_0 \rightarrow \mathbf{a}\ U_1 \\
& U_1 \rightarrow \mathbf{b}\ U_2 \\
& U_2 \rightarrow \mathbf{b}\ U_3 \\
& U_3 \rightarrow \varepsilon\}
\end{aligned}
$$

Dynamic Parsers

Suppose that s is a string generated by an evolving grammar with at least one grammar growth. The last step in the evolved derivation might be written as:

$$\beta w \underset{\Delta G^o}{\overset{R}{\Longrightarrow}} \alpha\, t_b w \underset{G^o}{\overset{R^*}{\Longrightarrow}} t_a t_b w = s^0 w = s$$

Where t_a, t_b, w, $s^o \in V_t{}^*$ and $\alpha, \beta \in (V_t \approx V_n{}^o)^*$. The parser will reduce the substring s^0 using the rules of the grammar G^0 only. Therefore the substring s^0 could be reduced by a conventional LR parser (P^0) implementing G^0. After doing the last reduction the parser has β on the stack and w (the string to be read) in the buffer . At that moment the grammar changes. Note that the grammar never changes on terminal shifting. The parser should then follow the derivation

$$\delta u \xRightarrow[\Delta G^1]{R} \gamma t_d u \xRightarrow[G^1]{R^*} \beta t_c t_d u = \beta s^1 u = \beta w$$

doing reductions in G^1 and scanning the substring s^1 and so forth for G^2, G^3... and s^2, s^3.... An evolving parser may be build by a conventional LR parser able to change its parsing tables at parsing time preserving the items on the stack. The input string may be split in several substrings s^i each one reduced using the grammar G^i only.

Note that the growth of grammars will usually produce a number of non active non-terminals and unreachable symbols that could be converted into active non-terminals and reachable terminals by subsequent evolutionary steps. Parsers designed to follow the growth of a growing grammar mustn't eliminate from their parsing tables non active non-terminals and unreachable symbols.

Towards real life evolving programming languages

We are going to introduce, in the following sections, a number of Z_k evolving grammars that will be progressively enhanced to create a starting grammar G_k^0,"good enough" for real life programming languages.

We will call this ideal "target evolving grammar" Z_z, and the intermediate steps toward our target starting grammar G_a^0, G_b^0, ...

Let us introduce an evolving language Z_aLanguage similar (in the beginning) to the metalanguage usually used to describe context-free production rules. The starting grammar of this language is G_a^0.

Suppose that the language $L(G_a^0)$ generated by the starting grammar G_a^0 allows the generation of sequences of 'statements' terminated by ';' and the only good 'statement' is a production rule.
A good string $s^0 \in L(G_a^0)$ could be s^0 = `statement->"mickey";` where the -> symbol would separate the two sides of a production rule, statement is a non-terminal symbol and "mickey" is terminal.

After parsing, the parser P_a^0 will change its tables according to the grammar G^1, where

$$G^1 = G_a^0 + \Delta G^0$$

and

$$\Delta G^0 = (\Delta V_n^0, \Delta \Phi^0) = (\text{statement, statement} \rightarrow \textbf{mickey}).$$

Then the string s^0 = `statement->"mickey";` itself is able to drive one step of grammar evolution. The more complex string $u = s^0 s^1$ where s^1 = `mickey;` is therefore a good string in the evolving language Z_aLanguage.

Actions on Production Rule Reduction

Let us now introduce an evolving language Z_bLanguage similar to the previous one plus some extension useful to define "<u>actions</u>" to be performed by the growing parser on application of production rules during the parsing phase. We will temporarily restrict the class of possible actions to additional growth steps to be performed on reduction of the rule to which the action is attached.

In other words Z_bLanguage admits, after a production rule, another production rule, enclosed by braces "{" , "}".
Now a good string $s^0 \in L(G_b^0)$ could be:

s^0 = `statement->"define""mickey"{statement->"mickey";};`

As in the previous example, when the parser reduces the last rule, it changes its tables according to the grammar G^1, where

$G^1 = G_b^0 + \Delta G^0$; ΔG^0= (statement, statement -> **define mickey**)

the parser with the new tables is able to continue the parse reading

s^1 = `define mickey;`

because s^0s^1 is a good string of Z_bLanguage.

After reading s^1 the P^1 parser generates a new evolutionary step

$G^2 = G^1 + \Delta G^1$; ΔG^1= (statement, statement -> **mickey**)

This happens because the "meaning" of the braces "{" , "}" in the language Z_bLanguage is to attach a ΔG to a rule. The string $u = s^0s^1s^2$

s^0 = `statement->"define""mickey"{statement->"mickey";};`
s^1 = `define mickey;`
s^2 = `mickey;`

would be accepted by the parser Z_bParser. Informally speaking, in this example we add a new statement "define mickey" which in turn is able to cause a further syntax change.

Up to now we stated that, on production rule reduction, a dynamic parser should be able to produce the grammar growth steps described inside the braces "{" and "}". It is useful now to introduce some more conventional classes of actions that should be performed by a dynamic parser on rule reduction and that could also be placed inside the braces.

A second class of actions will be the class of conventional semantic actions (i.e., the conventional semantic actions of the translation grammars). The dynamic parser should be able to call a number of "semantic routines", for example on the purpose of "code generation". To make realistic examples let us assume to have a predefined semantic action named "print" (and a related syntactical interface) capable of printing integer numbers and identifiers.
We suppose also that the syntactic rules for parsing identifiers (which we will identify by the non-terminal "ident") and integer numbers (non-terminal "num") are predefined in each Z_iParser, as well as some lexical analysis capability needed to separate the token of the parsed string (the token separators will be the blank and all the non alphanumerical characters).

Let us introduce now a third possible class of actions to be performed on rule reduction: the "return" actions (whose task is similar to the synthesis of the attributes of attributed translation grammars[C1]). Suppose now to parse the first statement of a program written in a new evolving language Z_cLanguage that admits, after a production rule, simple actions of the three types just now described (ΔG actions, semantic actions, return of synthetized attributes actions) enclosed by braces "{" , "}".

A good 'statement' of Z_cLanguage is

s^0 = `color->"red"{return 800 as num;};`

The desired "meaning" of this statement is: introduce a new grammar rule

ΔG^0 = (color, color -> **red**)

and synthetize, on reduction of "color -> **red**" an object of syntactic class "num" and of semantic value "800".

If the dynamic parser is able to do this job, then the meaning of the source statement s^1 ($s^0 s^1$ Z_cLanguage):

s^1=`statement->"wavelength of"color^wl"?"{print wl;print nanometer;};`

is: perform a grammar growth

ΔG^1=({statement, color},statement->**wavelength of** color **?**),

but remember that, on reduction of the rule 'statement->**wavelength of** color **?**' , two "print" semantic action should be executed. The first print uses for "wl" the semantic value of the synthesised attribute made available by the action "return" performed after reduction of the rule pertaining to the non -terminal "color". The second print is for the "nanometer" string. Now the last statement

s^2 = `wavelength of red ?;`

is parsed following the grammar $G^2 = G^0 + \Delta G^0 + \Delta G^1$ (a complete parse tree may be constructed), and two actions are performed: the first one will be the return of a synthetized attribute of syntactic class "num" and semantic value 800 after reduction of the "color -> **red**" rule and the second one will be the printing of the answer "800 nanometer". The execution of the semantic actions corresponding to the "print wl" and "print nanometer" actions can be executed using the semantic values "800" and "nanometer". Note that seemingly the "print wl" would be a print identifier statement, because at first sight wl is not a good "num". But as we have said the first action performed was to return an object whose syntactic class is "num" and whose semantic value is 800. To clarify how this mechanism can be effectively treated we are going to introduce some "volatile grammar changes".

Volatile Grammar Changes

The problem of correctly parsing the "action" can be formalized introducing "volatile grammar changes". Suppose that on parsing a segment of the form `name1^name2` (in our example `color^wl` in `statement->"wave length of" color^wl"?" {print wl; print nanometer;}`), the dynamic parser was able to save the following directive:

Before parsing the action `print wl; print nanometer;`

to be performed on reduction of statement -> **wavelength of** color^wl **?**
execute a volatile grammar change δG=(<type>, <type> -> **wl**)
where <type> is the type of the returned wl.

using wl as a temporary new terminal **wl**, and postponing the choice of the non-terminal to be used as <type> to the moment of reduction of the rule pertaining to the non-terminal "color" .

For example the parsing of the statement `wavelenght of red?`
would force the execution of the attribute synthesis action {return **800** as num}
attached to the rule color -> **red**
and before parsing the semantic action `{print wl; print nanometer;}`
attached to the rule statement -> **wavelength of** color **?**
the dynamic parser will activate a volatile grammar change δG = (num, num->**wl**)
and will use a synthetized attribute of syntactic class "num" and semantic value "800" on each occurrence of the volatile terminal **wl** within the action parsed with the volatile grammar.

Therefore the grammar used to parse the action will be

$G_C^{0+}(\Delta+\delta)G$ where
$(\Delta+\delta)G = \{(\Delta+\delta)V_n,\ (\Delta+\delta)\,\Phi\}$ with
$(\Delta+\delta)V_n =$ {statement, color, num}
$(\Delta+\delta)\,\Phi =$ {statement -> **wave length of** color **?**, color -> **red**, num -> **wl**}

In the next two sections we will outline an example of grammar growth driven by a simple source program. The example shows a growing grammar that dynamically incorporates new data types, declarations of variables, and overloaded operators.

An Example

This example shows a simple evolving language program using type identifier association, operator overloading and data type checking. The source program is :

```
statement -> integer ident^name {int_name -> name;};
statement -> real ident^name {real_name -> name;};
statement -> int_name^a "=" int_name^b "+" int_name^c;
statement -> real_name^a "=" real_name^b "+" real_name^c;

integer mickey;
integer donald;
real tom;
real jerry;

mickey = mickey + donald;
tom = tom + jerry;
```

The total grammar growth $G^8=G_C^0+\Delta G^{0:8}$ will be summarized by

$\Delta V_n^{0:8} =$ {statement, ident, int_name, real_name}
$\Delta\Phi^{0:8} =$ {statement -> **integer** ident, statement -> **real** ident,
int_name -> **mickey**, int_name -> **donald**,
real_name -> **tom**, real_name -> **jerry**,
statement -> int_name **=** int_name **+** int_name,
statement -> real_name **=** real_name **+** real_name}

Note that the "=" and "+" operators are now syntactically overloaded and that the syntactic coherence of the data types employed in the assignment and addition operations is strongly checked by the context-free grammar G^8.

In fact after the $\Delta G^{0:8}$ evolution the string `tom=mickey+jerry` will not be accepted.

Conclusions and Future Works

The dynamic parser "Z_ZParser" has been successfully employed to develop the evolving "APESE" programming language and other system software tools (e.g. the debugger) for the 100 GigaFlops SIMD parallel machine under development. The actual G_Z^0 grammar is a bit more complex than that used in the examples, to make the resulting "Z_ZLanguage" a friendly programming environment.

User programs written in the evolving Apese language are now running on APE100 parallel super computers. These programs, written using the evolving language, benefit of strong syntactic coherence checks and are several times shorter than programs written in conventional programming languages. These factors enhance the quality of the resulting application programs and shorten application development time. On the other hand dynamic compilations are quite slow. The compilation time of a program written in Apese language, generating 100,000 assembler lines, is 100 seconds on a Sun Sparc Station 2.

We got acquainted with the approaches of Burshteyn and Christiansen in 1992, at the end of the preparation of this article. We note, however, that using our evolving grammar approach, the language used to specify grammar evolution and residual semantic actions can be the evolved language itself. The user can then introduce new syntactic operators and perform semantic actions using a bootstrap procedure supported by the previously defined syntax.

The current conceptual framework shows several limitations in the area of "negative grammar changes". Therefore we hardly face problems connected with syntactic scope. We are working to extend the current scheme to catch negative changes too. We are also working to improve parser performances. For instance we are trying to balance the time spent by the Z_ZParser in parsing table changes and in conventional parsing phases.

Acknowledgements

Without the environment created by the INFN Ape100 group our work would never have come to existence. We owe a special debt to Walter Tross who helped us with comments and criticism and to Carlo Rovelli who took part in several preliminary discussions about this subject.

Bibliography

[G1] Burshteyn, B., "On the modification of the formal grammar at parse time", SIGPLAN Notices, Vol. 25, No. 5, pp.117-123, (1990)

[G2] Christiansen, H., "A survey of adaptable grammars",SIGPLAN Notices, Vol. 25, No. 11, pp.35 -44, (1990)

[G3] Burshteyn, B., "Generation and recognition of formal languages by modifiable grammars", SIGPLAN Notices, Vol. 25, No. 12 pp.45-52, (1990)

[G4] Heering, J., Kint, P., Rekers, J., "Incremental Generation of Parsers", SIGPLAN Notices, Vol. 24, No. 7, pp. 179-191, (1989)

[L1] Hopcroft, J.E. and Ullman, J.D.: "Formal Languages and their relation to automata", Mass., Addison - Wesley (1969)

[C1] Tremblay, J.P. and Sorenson, P.G.: "The theory and practice of compiler writing" Mc-Graw Hill (1987)

[C2] Aho, A.V., Sethi, R. and Ullman, J.D.: "Compilers Principles, Techniques, and Tools" Addison - Wesley (1986)

[Z1]"Apese Language" Ape100 Group Documentation A100/APESE/S -03 (1990), Phys. Dep. Univ. Roma "La Sapienza" Roma, Italy

[Z2] Cabasino, S., Paolucci, P.S. and Todesco, G.M., "Z_Z Language" Ape100 Group Documentation A100/ZZ/S-04 (1991), Phys. Dep. Univ. Roma "La Sapienza" Roma, Italy

[Z3] Cabasino, S., Paolucci, P.S., Todesco, G.M. "Dynamic Parsers, Evolving Grammars and Incremental Languages", Report N.863, Phys. Dep. Univ. Roma "La Sapienza" Roma, Italy,(1992)

[A1] Avico, N. et al., the Ape100 Collaboration: "From Ape to Ape100: From 1 to 100 Gigaflops in lattice gauge theory simulations", Computer Phys. Comm. 57 (1989), pp. 285-289

[A2] The Ape100 Collaboration: "A 100 Gigaflops parallel computer" Internal Note 733 Phys. Dep. Univ. Roma "La Sapienza" (1990) Roma, Italy

[A3] Tross, W. for the Ape100 Collaboration: "Status of the Ape100 project" Nuclear physics B (Proc. Suppl.) 20 (1991) pp. 138-140

Appendix A. G_C^0 starting grammar

$G_C^0 = (V_n^{\,0}, V_t, \Phi^0, \text{root})$ where

$V_n^{\,0}$ = {root, statement, thread, bead, action, name, number, string}

$V_t^{\,0}$ = {**->** , **{** , **}**, **ε** , ^ ,
identifier_token, **numerical_token**, **quotedstring_token**}

Φ^0 = {root **->** root statement, root -> statement,

statement -> **ident** **->** thread **{**action**}**,
thread -> thread bead, thread -> **ε**,

bead -> name^name,
bead -> name, bead -> number, bead -> string,

action -> action **any**, action -> {action}, action -> **ε**,

statement -> **return** name **as** name
statement -> **return** number **as** name,
statement -> **return** string **as** name,

statement -> **print** name,
statement -> **print** number,
statement -> **print** string,

name -> **identifier_token**,
number -> **numerical_token**,
string **->** **quotedstring_token**,

Rules enforced by the lexical analysis parsing phase

identifier_token -> any sequence of alphanumerical characters beginning with an alphabetic character,
numerical_token -> any sequence of numerical characters,
quotedstring_token **->** any sequence of characters enclosed by quotes,
any -> any sequence of characters followed by a '}'
}

Appendix B: Notation.

Closure Set V*

Given a set V the closure set of V, denoted as V^*, is defined as

$$V^* = \{\varepsilon\} \cup V \cup V^2 \cup V^3 \cup \ldots$$

where V^m designates all strings of length m composed by symbols in V and ε is the null string.

Metalanguage

A system or a language that describes the structure of another language is called a metalanguage.

Grammars $G=(V_n,V_t,\Phi,S)$

Grammars are metalanguages. A grammar is a 4-tuple $G=(V_n,V_t,\Phi,S)$ where:
- V_t is a finite non empty set of symbols called the terminal alphabet; the symbols in V_t are called terminal symbols;
- V_n is a finite non empty set of symbols called non -terminals (they are used in Φ to describe the syntactic structure);
- Φ is the finite non-empty set of "production rules", i.e., relations $\alpha \to \beta$ where

  $\alpha \in (V_t \cup V_n)^* V_n (V_t \cup V_n)^*$ and $\beta \in (V_t \cup V_n)^*$
- S is a distinguished element of V_n called starting symbol.

Direct Derivative $\psi \Longrightarrow \sigma$

For $\sigma \in (V_t \cup V_n)^*$, $\psi \in V_n^*$ σ is said to be a direct derivation of ψ, written as $\psi \Longrightarrow \sigma$, if there are strings φ_0 and φ_1 (including possibly empty strings) such that:
- $\alpha \to \beta$ is one of the production rules of Φ
- $\psi = \varphi_0\, \alpha\, \varphi_1$
- $\sigma = \varphi_0\, \beta\, \varphi_1$.

Reductions and Productions $\psi \overset{*}{\Longrightarrow} \sigma$

The string σ reduces to ψ (or ψ produces σ or $\psi \overset{*}{\Longrightarrow} \sigma$) if there are strings $\phi_0,\ldots\phi_n$

(n >= 0) such that $\psi = \phi_0 \Rightarrow \phi_1, \ldots, \phi_{n-1} \Rightarrow \phi_n = \sigma$.

Languages L(G)

The language generated by a grammar G is the set of strings σ such that

$$L(G) = \{ \sigma \mid S \stackrel{*}{\Longrightarrow} \sigma \text{ and } \sigma \in V_t^* \}$$

Rightmost Derivation and Ambiguous Grammars

Given a grammar G whose starting symbol is S and an input string x the rightmost derivation for x is given by

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{m-1} \Rightarrow \alpha_m = x$$

where the rightmost non-terminal in each a $_i$, is the one selected to be rewritten. A grammar is ambiguous if there is some string σ in the language that can be produced through different rightmost derivations.

Context-Free and Context Sensitive Grammars

A context-free grammar contains only production rules of the form
$\alpha \rightarrow \beta$, where $\alpha \in V_n$ and $|\alpha| <= |\beta|$, and $|\alpha|$ denotes the length of α.
A context-sensitive grammar contains only production rules of the form
$\alpha \rightarrow \beta$, where $|\alpha| <= |\beta|$.

LR(k) Grammars and Parsers

The LR class of grammars is essentially the set of all unambiguous context-free grammars. LR(k) parsers base their decisions using a parse stack and looking ahead the next k symbols in the input string. An LR(k) parser scans the input string form left to right constructing the reverse of the appropriate rightmost derivation.

Active Non-terminals

If a non-terminal symbol generates at least one terminal string of the language L(G), such a symbol is said active non-terminal.

Reachable Symbols

A symbol $A \in (V_t \cup V_n)$ which belongs to the set $\{A \mid S \stackrel{*}{\Longrightarrow} \varphi_0 \, A \, \varphi_1\}$.