Number-Crunching Software and the Input Problem: Guidelines and a Case Study

---

STEFAN GERBER AND HELMAR BURKHART

Institut für Informatik, University of Basel, Mittlere Strasse 142, CH-4056 Basel, Switzerland

ABSTRACT

Most of the number-crunching software running on supercomputers lack a concept for an error-free input processing. The programs often terminate after several hours due to user input errors. We present an analysis of this input problem. First, we separate the input parsing system from the number-crunching code. Second, we define a general grammar suitable for an automatic generation of a parsing system using the LEX/YACC tools. This concept leads to an input system that is easy to use and the generated input data for the number-crunching software is as error free as possible. We discuss the implementation of such an input system for an ab initio quantum chemical package.

© 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

The traditional number-crunching fields (e.g., quantum chemistry, molecular dynamics, and fluid dynamics) are the dominating CPU time consumers and use much of the available CPU resources of a supercomputing center. Improving the system performance does not change the situation as the users only adjust their calculation problems. The improvement of the software correctness or the correctness of the supplied input data can substantially save CPU time and money as the economic equivalent of a CPU-supercomputer hour is likely to be more than 1,000 US\$ [1].

Taking the quantum chemistry field as a prototype, programs like Gaussian [2] are very large (100,000 lines of Fortran code) and date mostly

from the 1970s. Such programs have grown like an amoeba because quantum chemistry has developed towards new fields, better algorithms have been invented and will be developed in future, and the existing code has been modified and enhanced over the decades. As a result, the input (not speaking of the other parts) to these archaic monsters is very complex and in most cases context sensitive as different data have to be supplied for different tasks. Most of these programs require formatted input data (2I4,F10.7, . . .); however, input to more recent programs has improved towards free-format input but with many exceptions ("give one and only one title line after the first command line" [3]). Software ergonomics has proved that such user-unfriendly interfaces produce many errors. The situation is even worsened by the batch-oriented character of the programs. As the read statements for the input are normally distributed over the whole program, it often happens that an input failure is detected at a very late state of the calculation after several hours of wasted CPU time.

Beside the correctness of the supplied input data, further problems are the extendibility and

---

Received August 1992
Accepted December 1993

© 1994 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 3, pp. 1–11 (1994)

CCC 1058-9244/94/010001-11

portability of the programs. No research group has the manpower to rewrite 100,000 lines of Fortran code grown over decades into a more suitable and structured program. Practically all current implementations of input modules with free-format input are heavily computer dependent because the underlying parsing system needs pointer types that are not standard Fortran data types. Sometimes the code is even compiler dependent. However, the ability to use faster machines to get more CPU time is a credo of the number-crunching community and this brings portability into view. Besides the demand of having a very fast and normally machine-dependent block I/O facility, the second obstacle for portability is located in the input modules. So, portability is another reason for having an easily usable and extendible input program at hand.

In Section 2 we give the guidelines for mastering the input problem. By following these demands it is possible to implement an input program that delivers the number-crunching program with correct data. In Section 3 we discuss the basic elements of the input language and the general grammar in terms of extended Backus-Naur Forms (EBNF). We give a concrete example of a quantum chemical input language and we discuss and show the worth of such a language. Section 4 covers implementation aspects of an input language. The data structure required, the data manipulation routines, and the production rules of the parser are discussed in detail. The input program *Idefix*, which delivers our *ab initio* quantum chemical package with proper input data, serves as a case study. Finally we draw the conclusions in the last section.

2 GUIDELINES

The quest for high-performance computing systems for solving real-world problems leads to a waste of CPU time if no precaution is taken against incorrect input data. To benefit the scientific community, concepts for mastering this input problem are needed. This section gives a list of postulates that should allow the input problem to be overcome:

1. *Input parsing and number-crunching parts are separated into independent units:* This demand frees the programmer from using the historically based Fortran language for the parser modules. An adequate language

can be chosen to build the input module and the number-crunching software then reads the generated input from file by standard Fortran read statements. It is almost always better to do checking at input, and try to separate the checking part from the computation part to achieve more data independence. The benefit is enhanced portability and reliability of the number-crunching code. Furthermore, the problem of input data correctness is shifted to the input module. However, as this module is not time critical, the economic usage of CPU time is highly warranted if the input module serves the number-crunching module with the generated data.

2. *Encapsulation and naming of data by the concept of logical blocks:* Despite the mandatory input data for a certain program, there is a plethora of often optional program-specific parameters that are responsible for the flow of control. To structure these data, we define a grammar with logical blocks for different types of information. The logical block itself can consist of further logical blocks and data. In quantum chemistry programs, for example, the mandatory requirements are the space coordinates of each atom that participates in the molecule of interest. Second, a set of basis functions for each atom must be supplied. Each atom now builds a logical block with two sub-blocks for the space coordinates and the basis functions, respectively. On the number-crunching side each block of data corresponds to a subroutine capable of reading these data. The parsing system transforms the data to a correct format known by these routines; so these data can be read in easily. The number-crunching parts benefit from the structured data as no parsing system is needed and portability is guaranteed.
3. *Reliable error detection:* Two types of errors are possible: syntax errors that are detected by the parsing system almost for free and logical errors. For instance, in quantum chemistry we usually demand a certain accuracy for the integral evaluation. The accuracy can be sufficient for a self-consistent field (SCF) calculation, but might be too low for a many-body perturbation approach. If the accuracy is too low, the calculated results are meaningless because of the un-

certainty due to the numeric. Furthermore, forcing an encapsulation into logical blocks, the logical error check can be accomplished as soon as the token closing the logical block is seen by the parser. All the information of the block is then available and can be checked.

4. *Context-free, case-insensitive, and free-format input grammar:* To avoid such boring statements in the reference manual like "If a symmetry group directive is included it must come before the geometry specification" [3], we demand a context-free grammar, which means that the definition of a logical block has no influence on the definition of any other block. The user can define the input data in any desired order. To extend the input quality, the supplied data are free format and case insensitive (e.g., keywords or logical block names). This is the most important demand in the user's point of view, as most errors are trivial (incorrectly spelled or the given format is not correct), caused only by exceptions from the general input grammar rules.
5. *Ask for a minimum amount of user input:* In order to decrease the error rate, reasonable defaults for the parameters are set. The intention is to define the parameters for a standard type of calculation that should be the least time critical. For instance, in quantum chemistry one would apply the SCF options ($O(n^4)$) as default because a post-SCF type of calculation ($O(n^5)$ - $O(n^7)$) is substantially more expensive. Nevertheless, defaults can be overridden at any point of time by specific user-supplied data.
6. *Concept of macro blocks:* To further structure and to shorten the user input, the capability of defining a macro name for a whole logical block must be built in. Later on, the whole block can be referenced by that macro name. This conception ensures the reusability and correctness of the user-supplied data. In quantum chemistry the definition of a basis set for a certain atom can consist of up to 100 real numbers (at least seven digits precision). If the molecule of interest has more than one atom of a certain type, the user defines the basis once and references the macro name each time the basis is needed.
7. *Predefined macros in a user-extendible*

text library: The idea of macro definitions is further extended such that a macro definition used for more than one input file can be put into a text library. Each time a macro name is referenced, the internal list of macros (defined in the current input stream) is scanned and thereafter the text library is searched for that macro name. The text library is just a collection of logical blocks defined in the correct grammar that are tagged with a macro name.

If the input parser system fulfils the guidelines given above, the input produced, which is suitable for the number-crunching part, should be optimally error free. The parser system is easily extendible as there are no cross-connections between the logical blocks. A new type of input information can be directly mapped to a new logical block, and on the number-crunching side the new information is read in by a new input routine.

3 THE LAYOUT OF THE INPUT GRAMMAR

In this section we define the syntax rules for the input. The basic vocabulary consists of symbols classified into letters, digits, and special symbols. **Identifiers** are names denoting macro names or words like a file name. They must begin with a letter followed by any combination and number of letters and digits. The class **<keyword>** represents a set of reserved identifiers. The user may not use these words in a context other than that explicitly imposed by the input language, i.e., these words are not legal identifiers (or macro names). Furthermore, the set of keywords is primarily given due to the nature of the number-crunching program. For instance, the chemical symbols of the periodic table are keywords for a quantum chemical program, whereas there is no need to define these symbols for a fluid dynamic program. Decimal notation is used for **numbers**. Sequences of any characters enclosed by single quotation marks are called **strings**. Blanks, tabs, and EOLs are considered as separators. Any number of separators may occur between two consecutive symbols. However, at least one separator must occur between any pair of consecutive identifiers, numbers, or keywords. The general production rules for an input language are now given in terms of EBNF.

```

<input> ::= <macro declaration><logical block>
<macro declaration> ::= {<macro name> = <logical block>}
<macro name> ::= <identifier>
<logical block> ::= <blockname> = <macro name>
                  | <blockname> ({<logical block>|<logical block def>})1
<logical block def> ::= <keyword> = <value>|<boolean>|<identifier>|<string>
<blockname> ::= <keyword>
<keyword> ::= <identifier>
<value> ::= <number> [<unit>]
<unit> ::= <keyword>
<boolean> ::= on|off

```

As explained in Section 2, we structure the input by collecting a certain type of data into a logical block. Consequently, several logical blocks can be controlled by a supervisor block. This concept of nesting and hierarchically grouping types of data demands that the scope or range of validity of a keyword is limited. By limiting the scope, a keyword can also be used in different blocks with different meanings. Often naming conflicts result when defining commonly used expressions of a number-crunching field as keywords. For instance, the chemical symbol of sulfur and the total symmetric orbital type are both abbreviated by "S". However, as the atomic and orbital type information is used in different blocks and the scope of the keyword is limited to a certain block, the lexical analyser can easily distinguish between the different meanings. One can think of the block name (keyword) tagged to a logical block as a transformation function that reduces the whole set of keywords to a subgroup of recognizable keywords inside the currently opened block. This mechanism of focusing the set of keywords ensures a proper nesting of logical blocks and thus a proper structuring of data.

3.1 Case Study of an Input Language

To give an impression of how an input example looks, we use Idefix, our input generator for the ab initio package Obelix, as a demonstrating platform. The grammar used (see the technical report [4] for the full grammar) is an explicit instance of the general grammar outlined in the last paragraph. As quantum chemistry deals with molecules, it is a natural choice to structure the input as shown in Figure 1.

The mandatory definitions for a molecule are the set of participating atoms. Each atom in turn consists of a set of space coordinates and a so-called basis set. The basis set is further modeled by different orbital types (s,p,d, . . . angular mo-

mentum functions), and each orbital type is then constructed by a set of so-called exponents α and associated coefficients c . These are parameters for the Gaussian lobe functions [5, 6] used in our ab initio package.

The input for the simplest molecule H_2 applying the simplest level of theory (SCF) has the layout shown in Table 1 (note that the case of the letters and the indentation is made only for better visualization of the block structure and has no influence on the input meaning; underlined words are keywords).

In line 1 the basis set is defined as a macro consisting of a simple s type orbital with one exponent $\alpha = 0.71367655$ and a coefficient $c = 1.0$. The basis block is then assigned to the macro name *theHBasis*. Inside the atom blocks the basis is directly referenced by that macro name (lines 3 and 4). The first atom block (line 3) is opened by the chemical element name, whereas for the second atom block (line 4) the chemical element symbol is used. Both keywords are legal identifiers for the same type of nucleus. The location of the first nucleus is in the origin of a cartesian coordinate system, whereas the second hydrogen is placed on the z axis 74.144 pm away from the origin. The

FIGURE 1 Logical layout of an ab initio input.

Table 1. Primary Input Definition for the H₂ Molecule Example

---

```

theHBasis = Basis(s(exp = 0.71367655 coeff = 1.))
Molecule(
    Hydrogen( Cartesian(x = 0. y = 0. z = 0.) BASIS = TheHBasis)
    H( Cartesian(y = 0. z = 74.144 x = 0.) Basis = theHbasis)
)

```

---

length unit pm is the default length unit for any metric input and therefore may be unspecified. Atomic units (au) and Ångstrom (Å), which are widely used in the quantum chemistry area, are the other legal metric units but must be explicitly set.

Note that all other blocks beside the molecule block and its descendants are optional and their parameters controlling the flow of control will be preset with reasonable defaults. Naturally we can override each of these parameters. For instance, we can give a title to describe the type of calculation:

TITLE = 'SCF test run for H2 with 1s type basis function on each nucleus'

Given the input file, Idefix generates an output file that is suitable for the ab initio package Obelix. Table 2 shows the output produced for the H₂ example. Note that the bond length of 74.144 pm has been converted to atomic units due to the requirements of the ab initio program. It is an obvious benefit to get a properly formatted one page input file for the ab initio package out of five lines of primary input. In addition, parameters for the control flow are automatically included and preset with reasonable defaults.

Features of the ab initio package can be selectively turned on or off, e.g., the statement

Properties(Population = on Multipole = on)

activates the Mullikan population analysis and the multipole analysis. To make the input definition even easier and correct, we have defined an archive of basis sets. This archive is user extendible and serves as a pool of reusable basis set modules. A specific basis set is loaded by the command

Library = *name*

where *name* is such a predefined and stored basis set. Note that chemically interesting, medium-sized molecules consist of 10 or more atoms (e.g.,

benzene C₆H₆). Applying reasonable large basis sets to each atom (e.g., Dunning's 10s6p basis set [7], Idefix generates 389 lines out of 15 lines of primary input! So, the benefit is even more striking as the molecule of interest grows.

4 IMPLEMENTATION ASPECTS

Because of the widespread availability of UNIX platforms and the programming language C we have decided to use C [8] in combination with the translator-writing tools LEX [9] and YACC [10] to build automatically the scanner and parser. These tools generate C source code, so that we can even port the scanner and parser to systems where the translator-writing facilities are not available, e.g., VAX/VMS systems. By using the LEX/YACC pair most of our basic requirements are automatically fulfilled. The LEX grammar for Idefix consists of 238 lines, and the YACC grammar for Idefix has 424 lines, whereas the two generated C source codes together are 127 kB of C source code. If a translator-writing tool is used for building a programming language, a mechanism must be provided that handles the identifier table, the nesting of blocks, the allocation of descriptor blocks, etc. The same is true for an input generator. We find the notion of block structures and thus need semantic routines for allocating blocks, defining block entries, and checking the block consistencies. Writing these semantic routines is the main implementation task. In the following we give a survey of Idefix, an implementation of an input generator for a quantum chemistry program. Idefix is fully designed within the demands postulated in the previous paragraphs. Each general topic is illustrated by an appropriate part of Idefix. A more detailed description of the implementation can be seen in the technical report [4].

4.1 Internal Data Structure

Each logical block is projected to a data structure of the general type:

Table 2. The Generated Input of the H₂ Molecule Suitable for the ab initio Package Obelix

---

```

**** Original input file: H2.COM
**** File H2.JOB from: Wed May 20 10:13:00 1992
**** Copyright for input compiler: S. Gerber, IFI, Univ. of Basel, 1991
**** DIMENSION_DATA
    2 1 2 2 1 4
**** FILE_INFO
H2.INT
H2.SAV
H2.DEN
DELETE DELETE FORMATTED
**** DEBUG
    0
**** SAVE
    2
    3 T
    1 T
**** PRINT
    1
    1 F F 1 -1
**** JOB_DATA
    1 F -1
**** MOLECULE_DATA
    0 1 2.0159

    H2
**** SLAVE_OPTIONS
    1 100
**** INTEG_OPTIONS
    1.000000e-09 0.0100 0.0300 0.0500 T T T T
    -1 50000 -1
**** SCF_OPTIONS
    1.000000e-06 30F
    T 6 0.4000
    F 0 0.0000
    F 0 0 0.0000
    F 0 0.0000 0.0000
    F 0.0000
**** MP2_OPTIONS
    1
    F 0 0 0
**** PROPERTIES
    F F F F
**** GUESS_INFO
    1 1 0 0 0 0
    1 1 0 0 0 0
**** ATOM 1
H 1 1.0080 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
**** BASIS OF ATOM 1
    1 1 0 0 0 0
    1.0000 No library basis used
    S 1 1.0000 1.0000
    0.7136765500e+00 1.0000000000e+00
**** ATOM 2
H 1 1.0080 0.0000000000e+00 0.0000000000e+00 1.4172951750e+00
**** BASIS OF ATOM 2
    2 1 0 0 0 0
    1.0000 No library basis used
    S 1 1.0000 1.0000
    0.7136765500e+00 1.0000000000e+00
**** TITLE
    1
'SCF testrun for H2 with 1 S type basis function on each nucleus'
*** End of File H2.JOB

```

These data are preset and included automatically

} Cartesian Coordinate

} Basis set of atom 1

} Cartesian Coordinates

} Basis set of atom 2

} Given title

```
typedef struct {    void *theSupervisor;
                   Datum entry for each type of block specific information
                   Pointer for each descendant of the current block
}BlockNameStruct, *BlockNamePtr;
```

the Supervisor variable always points to the next higher block in the hierarchy (*theSupervisor* of the topmost block is NIL). As an example we look at the atom block, a block holding information of one atom of the molecule of interest:

```
typedef struct {    void*          theSupervisor;
                   int           theOZ;
                   double        theNucleusMass;
                   int           theEFGProperty;
                   CoordinatePtr theCoordinatePtr;
                   BasisPtr      theBasisPtr;
} AtomStruct, *AtomPtr;
```

The supervisor of the atom block is the molecule block, which holds all atom blocks building the target molecule. The atomic number (*theOZ*), the mass of the atom, and the electric field property are data definition specific for that atom. *theCoordinatePtr* and *theBasisPtr* are links to corresponding blocks for the location in space and the basis on that atom. These references will be set as soon as these blocks are opened (see below).

4.2 Flow of Control

The general flow of control of the Idefix program is shown in Figure 2. The input data are interpreted by the scanner system, which in turn is controlled by the parser. The parser also has access to the reusable library of basis sets and it interacts as well with the semantic routines. Finally, the output generator produces an output file that is suitable for the specific needs of the number-crunching program.

4.3 The Semantic Routines

For each logical block, four semantic routines are needed for manipulating and managing the data flow. Their interaction with the parser is visualized in Figure 3.

InitBlockName   e.g., InitCartesian

The appropriate InitBlockName routine is called after the parser has detected a new blockname. The routine is responsible for allocating memory for that block and for initializing a default value for each datum of that block. The last action is to

call the AdjustContext routine with the block name as argument. This routine in turn adjusts the lexical analyzer context to the set of allowed keywords for the current block (for a description of AdjustContext see Section 4.4).

AddToBlockName   e.g., AddToAtom

The AddToBlockName routine is responsible for connecting the new structure allocated by InitBlockName to the appropriate field in the block of the supervisor. This means for example that the

FIGURE 2 Coarse grain flow of control of the Idefix input generator.

FIGURE 3 Interaction between parser and semantic routines.

AddToAtom routine does connect descendant blocks of the atom block (e.g., the basis block or the coordinate block) to the atom block whereas the atom block is connected by the AddToBlockName routine of the supervisor. The routine then assigns the address of the new structure to a global variable that holds the current block in use. Now the context of the parser and the semantic routines is fully switched to the new logical block.

SetBlockNameDatum

e.g., SetCartesianDatum

SetBlockNameDatum is used to store any datum into a field of the structure of a logical block. Because the datum can be of type string, integer, or float, three arguments (one for each type) are passed to SetBlockNameDatum. The argument type of interest is used inside the routine and the others are ignored. A keyword tag is given as the first argument to signify which datum to set in the structure.

CheckBlockName e.g., CheckCartesian

The CheckBlockName routine tests the validity of the data and the existence of descendant blocks if any (note that the validity of data of descendant blocks is checked by the check routine of the descendant block itself). The type of test is heavily

dependent on the nature of the block. For instance, the validity of exponents and coefficients of the supplied basis set can only be tested for their existence, whereas the convergence accuracy must have a minimum value defined by the type of calculation. If any of the checks fails, an appropriate error message is generated and the program immediately terminates gently. The last and mandatory action of each check routine is to reset the global variable holding the address of the current block to the block address of the supervisor and to switch the lexical analyzer context to that of the supervisor.

4.4 The Scanner and Parser

The programmer does not have to care about the scanner as it is fully managed by the parser. The only direct manipulation of the scanner by the programmer is through the AdjustContext routine. This routine focuses the set of keywords to the active set of keywords through the concept of lexical start conditions. The name of a start condition is given in the definition section of the scanner file by %start blockname (e.g., %start MOLECULE). Each token (keyword recognized by the scanner) gets a start condition attribute by referencing the condition at the head of a token by {blockname} expression (e.g., {MOLECULE} {M}{O}{L}{E}

{C}{U}{L}{E}). When opening a new block the AdjustContext routine executes the action statement BEGIN blockname; (e.g., BEGIN MOLECULE;) to enter the scanner into the appropriate start condition. The AdjustContext is called inside each initialization and check routine and appears as:

```
AdjustContext (int BlockNameTag) {
    switch (BlockNameTag)
    {
        BlockNameTag for each block:
        BEGIN blockname;
    }
}
```

The handling of the various logical blocks in the parser is always the same, whether they are mandatory or not. For instance, when the keyword for an atom (chemical symbol <H> or name of an element <hydrogen>) is found the sequence (statements in curly brackets are C functions, the rest is YACC grammar)

```
ATOM { InitAtom(false); }
      ('AtomBlock') { CheckAtom(); }
```

is executed (ATOM is the blockname tag defined by the %token instruction). AtomBlock, which is between the initialization and the check of the block, then describes the grammar for that block. The YACC grammar for AtomBlock is defined as

```
AtomBlock:  AtomDef
           |  AtomBlock AtomDef
```

to ensure proper repetition. Each datum in the block is represented by a line similar to:

```
AtomDef: MASS '=' REAL
        {SetAtomDatum(MASS, 0, $3, NULL); }
```

where SetAtomDatum assigns the mass of the nucleus to the appropriate variable of the block. To allow macro assignments for a whole block a YACC grammar line of the type

```
ATOM '=' WORD {AddToMolecule(ATOM,
                             (AtomPtr) SearchMacro($3, ATOM)); }
```

is defined. The SearchMacro function searches the macro list for the macro name (WORD) and returns a copy of the logical block. The block is then connected to the block of the supervisor by the AddToMolecule routine.

4.5 The Output Generator

The parser system terminates its job when it recognizes the end of file EOF character and gives the control back to the main program, which in turn calls the output routine **WriteBlockName** of the topmost logical block. The duty of each write routine is to output all data of the current block in a format defined by the number-crunching program. Thereafter, the write routines of all descendant blocks, if any, are called:

```
WriteBlockName(BlockNamePtr theBlockNamePtr, FILE *theOutUnit) {
    DescendantBlockNamePtr, theDescendantBlockNamePtr;
    output all datum entry in the appropriate format
    theDescendantBlockNamePtr = theBlockNamePtr->theDescendantBlockNamePtr;
    WriteDescendantBlockName(theDescendantBlockNamePtr, theOutUnit);
}
```

The output routine for an atom block is therefore implemented for Idefix in the following way:

```
WriteAtom(AtomPtr theAtomPtr, FILE *theOutUnit) {
    fprintf(theOutUnit, "%2s%4d%10.4f", ChemSymbols[theAtomPtr->theOZ],
                                                    theAtomPtr->theOZ,
                                                    theAtomPtr->theNucleusMass);
    WriteCoordinate(theAtomPtr->theCoordinatePtr, theOutUnit);
    WriteBasis(theAtomPtr->theBasisPtr, theOutUnit);
}
```

4.6 Macro Definition Section

The same grammar sequence as in the normal definition section is carried out if a macro is defined. The only difference is that both the pointer to the logical block (*theBlockNamePtr*) and the macro name are stored in a macro list. If the macro name is later referenced, the search routine **SearchMacro** looks for the first occurrence of that macro name. The block is copied and the pointer to this block is connected to the block of the supervisor by the appropriate **AddToBlockName** routine.

4.7 Extensibility

One motivation for the development of Idefix was the necessity of having an easily extendible input system. As we are constantly tuning and extending our *ab initio* package Obelix, we must be able to control easily new classes of algorithms or new features of existing algorithms by incorporating the appropriate parameters into the input system. Two cases can occur: either new options must be built into an existing logical block or a new logical block must be defined.

Incorporating a New Datum Entry into an Existing Block

- Step 1. Define a new datum entry in the existing *BlockNameStruct* record.
- Step 2. Set a reasonable default value in the *InitBlockName* routine.
- Step 3. Update the *SetBlockNameDatum* routine in the way capable of assigning a value to the new datum entry.
- Step 4. Test the datum entry for a legal value range in the *CheckBlockName* routine.
- Step 5. Write the new datum entry onto the output file in the *WriteBlockName* routine with a format given by the number-crunching package.
- Step 6. Extend the scanner with keywords if necessary.
- Step 7. Define the new feature in terms of a correct YACC syntax and call the *SetBlockNameDatum* routine appropriately.

Define a New Logical Block

In the case of a new algorithm in the number-crunching code one often needs many options to

control the behavior of that algorithm. Consequently a new logical block must be defined with its own set of keywords. Six steps are needed to extend the input system towards controlling these new features.

- Step 1. Define a new record representing the logical block and create data entries for each option needed.
- Step 2. Write the three semantic routines, and the output routine for the logical block.
- Step 3. Define an entry for the new logical block in the block of the supervisor.
- Step 4. Extend the scanner and parser in the appropriate way by defining the new keywords and the correct syntax. Furthermore, add in the LEX grammar file a new start condition to the set of start conditions by the *%Start* instruction.
- Step 5. Adjust the *AdjustContext* routine so that the aggregate of all keywords can be transformed to the active set of keywords suitable for the new block.
- Step 6. Call the semantic routines inside the parser appropriately, and call the *WriteBlockName* routine inside the *WriteBlockName* routine of the supervisor.

5 CONCLUSIONS

Due to the "old style" input facilities of most number-crunching software, better concepts are required for the input processing part. We have discussed a new approach for mastering the input problem and an explicit implementation for an *ab initio* package. We have achieved the goals mentioned in Section 2 by:

1. Separating the input modules from the number-crunching parts. The correctness and reliability of input data delivered to the number-crunching modules are, as much as possible, ensured prior to running the time consuming program and thus we overcome a waste of computer time due to input failures.
2. The defined grammar suitable for all number-crunching areas is context free, case insensitive, and free format.
3. Portability of the parsing system can be guaranteed as we use LEX and YACC

translator-writing tools, which generate the appropriate scanner and parser, respectively.

4. To address extendibility we outlined a few steps a programmer has to follow for future enhancement of the input system.

Our experiences with the Idefix input compiler are quite encouraging as we have had no input failures since its use in the ab initio package. Getting started with the ab initio package is very easy even for newcomers in that area.

ACKNOWLEDGMENTS

The authors thank Dr. D. Searles for her help in translating this document into English. The research work was supported by the Swiss National Science Foundation under grant 21-25410-88.

REFERENCES

1. Private communication: Each CPU hour of the CRAY Y/MP at the Federal Institute of Technology ETH in Zürich is charged 3600 sFr ($\approx$ 2500\$).
2. M. J. Frisch, M. Head-Gordon, G. W. Trucks, J. B. Foresman, H. B. Schlegel, K. Raghavachari, M. A. Robb, J. S. Binkley, C. Gonzales, D. J. Defrees, D. J. Fox, R. A. Whiteside, R. Seeger, C. F. Melius, J. Baker, R. L. Martin, L. R. Kahn, J. J. P. Stewart, S. Topiol, and J. A. Pople, *Gaussian 90*. Pittsburgh, PA: Gaussian, Inc., 1990.
3. R. D. Amos and J. E. Rice, *CADPAC—The Cambridge Analytic Derivatives Package, Issue 4.0*. Cambridge: University of Cambridge, 1987.
4. S. Gerber, and H. Burkhardt, "Number-crunching software and the input problem: Guidelines and a case study," Institute für Informatik, Technical Report no. 93-4, Universität Basel, 1993.
5. S. F. Boys, "Electronic wavefunctions. A general method of calculation for the stationary states of any molecular system," *Proc. R. Soc. (Lond.)*, vol. A200, pp. 542–554, 1950.
6. J. L. Whitten, "Gaussian lobe function expansions of Hartree-Fock solutions for the first-row atoms and ethylene," *J. Chem. Phys.* vol. 44, pp. 359–364, 1966.
7. T. H. Dunning, "Gaussian basis functions for use in molecular calculations. III. Contraction of (10s6p) atomic basis sets for the first-row atoms," *J. Chem. Phys.*, vol. 55, pp. 716–723, 1971.
8. B. W. Kernighan and D. M. Richie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
9. M. E. Lesk and E. Schmidt, *Lex—A Lexical Analyzer Generator*. Murray Hill, NJ: Bell Laboratories, 1975.
10. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*. Murray Hill, NJ: Bell Laboratories, 1975.

Hindawi

Submit your manuscripts at
<http://www.hindawi.com>