EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

A SPICE input to NDML compiler

van Waes, R.J.J.

Award date:
1984

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Arie Hoddeler

5532

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Electrical Engineering
Research Group ES

A SPICE INPUT TO NDML COMPILER

R.J.J. van Waes, December 1984

Professor : Dr. Ing. J.A.G. Jess
Coach : Ir. G.L.J.M. Janssen

The Eindhoven University of Technology accepts no
responsibility with regard to the contents of this report.

SUMMARY

A piece-wise linear simulator for electronic circuits [1],[2] has been developed in the research group ES of the Eindhoven University of Technology. To be able to accept circuit descriptions prepared for the simulator SPICE [3] and to compare the performance of the piece-wise linear simulator and SPICE a compiler has been constructed. This compiler translates input for SPICE to input written in NDML (Network Description and Modelling Language) [4],[5] , which is the input language of the piece-wise linear simulator.

This report describes the design of the compiler and how it can be used.

CONTENTS

INTRODUCTION.. 5

1. THE SOURCE LANGUAGE : SPICE 2G INPUT................ 6

1.1 The Syntax of SPICE 2G Input................... 6

1.2 Semantics of SPICE 2G Input.................... 7

2. THE OBJECT LANGUAGE : NDML.......................... 9

2.1 The Syntax of NDML............................. 9

2.2 Semantics of NDML.............................. 9

3. THE COMPILER.. 11

4. LEXICAL ANALYSIS.................................... 12

4.1 The Lexical Analyser Generator Lex............. 12

4.2 SPICE Input.................................... 14

4.3 Start Conditions............................... 14

5. THE PARSER.. 17

5.1 Usage of Yacc.................................. 17

5.2 Implementation................................. 17

6. SEMANTICS... 18

6.1 Translating a Circuit Element.................. 18

6.2 Buffers Used for Temporary Storage............. 20

6.3 The Translation of Subcircuits................. 21

6.4 The Translation of Comment Cards............... 22

6.5 Cards That Are Not translated.................. 22

6.6 Leafcells Used................................. 23

6.7 The Contact List and the Wire List............ 23

6.8 The Translation of Numbers..................... 24

7. ERROR HANDLING.. 25

7.1 Errors Detected By The Lexical Analyser........ 25

7.2 Errors Detected During Parsing................. 25

8. USAGE.. 27

LITERATURE.. 31

APPENDIX 1: CONTEXT-FREE GRAMMARS AND BNF............... 33

APPENDIX 2: THE SYNTAX OF SPICE 2G INPUT................ 36

1. Higher Part...................................... 36

2. Subcircuit-calls................................. 37

3. Cards Beginning With a Period.................... 37

4. Resistor, Capacitor, (Coupled) Inductor.......... 38

5. Transmission-line................................ 40

6. Controlled Sources............................... 41

7. Independent Sources.............................. 42

8. Semiconductors................................... 45

9. Lexical Rules.................................... 48

APPENDIX 3: DESCRIPTION OF FUNCTIONS, VARIABLES AND

MACROS USED... 54

1. Functions Invoked by Main........................ 54

2. General Functions................................ 56

3. Error Handling................................... 59

4. Alphabetic List.................................. 62

INTRODUCTION

To be able to construct a compiler, one must know the syntax and semantics of both the source language and the object language. The source language, in our case the input language for SPICE, is discussed in chapter one. The object language, in our case the input language NDML for the piece-wise linear simulator, is discussed in chapter two.

The compiler itself consists of three parts : a lexical analyser, a parser and the semantic functions called by the parser. The chapters three to six deal with the compiler and its three parts.

The way errors are handled is described in chapter seven.

Chapter eight describes how to use the compiler.

Appendix three contains a description of the semantic functions, variables and macros used in the compiler. This may be helpful for future modifications of the compiler.

In this report no special typographical notation is used for names, characters, strings, expressions et cetera, except for appendix three, where the names of functions, variables and macros are written bold.

1. THE SOURCE LANGUAGE : SPICE 2G INPUT

SPICE is a general-purpose circuit simulation program for nonlinear dc, nonlinear transient, and linear ac analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, bipolar junction transistors (BJT), junction field effect transistors (JFET) and metal oxide semiconductor field effect transistors (MOSFET). SPICE has built-in models for the semiconductor devices, and the user only needs to specify the pertinent model parameter values.

SPICE can give the current and voltage at any point in the circuit, in table form and plotted.

1.1 The Syntax of SPICE 2G Input

Appendix 2 contains a formal definition in the extended Backus Naur form [6] (See appendix 1) of the syntax of SPICE 2G input. This definition has been derived from the rather informal and incomplete information given in the SPICE version 2G User's guide [3].

In practice SPICE 2G appears to allow more than stated in the User's guide:

- The asterisk mark * marking a comment card doesn't have to be placed in the first column. It may be preceded by any combination of separators.
- The plus + , indicating continuation of the previous card, may also be preceded by any combination of separators.
- Alphanumeric strings and names may contain the following symbols:

```
+}<>?!"#$%'-^@*]:;/_.
```

We have included these peculiarities in our syntax definition listed in appendix 2.

SPICE 2G considers its input as divided into a set of cards. The first card is a title card. The last card must be an END card, only filled with the string .END .

The other cards can be divided in the following groups:

- control cards , that define the model parameters and the run controls
- element cards , that define the circuit topology and element values

1.2 Semantics of SPICE 2G Input

In this section only a brief overview of the semantics are given. For further details see [2].

1.2.1 Control Cards

The control cards allow the user to specify

- what kind of analysis is wanted: DC analysis, AC analysis, transient analysis or analysis at different temperatures
- which currents and voltages are to be printed or plotted in the output
- where subcircuit definitions begin and end in the input
- the models used for semiconductors

Subcircuits may be nested. The SPICE version 2G User's Guide [3] states that there is no limit on the size or complexity of subcircuits and on the nesting level.

1.2.2 Element Cards

An element card contains:

- a reference to a specific kind of element (for instance a resistor, a capacitor or an inductor)
- the instance name of the element
- the nodes of the element, which indicate where the element is placed in the circuit
- the values of parameters such as resistance, temperature coefficients, initial conditions and the like.

2. THE OBJECT LANGUAGE : NDML

The Network Description and Modelling Language NDML is the input language for the piecewise linear simulator. The simulator is mainly intended to simulate large integrated electronic circuits, but in fact it can be used to simulate any system that can be described mathematically in piecewise linear models. See [1] and [2]. The simulator features mixed-mode mixed-level simulating. Mixed-mode means that both digital and analogue electronic circuits can be simulated. Mixed-level denotes that any level of the system can be simulated. AC small signal analysis of electronic circuits is not yet implemented in the simulator.

2.1 The Syntax of NDML

A formal definition in the extended Backus Naur form of the syntax of NDML is given in appendix 3 . The extended Backus Naur form is explained in appendix 1.

2.2 Semantics of NDML

A complete description of the semantics of NDML can be found in [4] and [5]. This section only points out the main features.

NDML supports a hierarchical structure of the circuit (also called system) to be defined. Therefore the terms leafcell and compound are introduced. Leafcells are the lowest components in the hierarchy. A compound contains one or more instances of leafcells and other compounds. These instances are called subsystems.

A leafcell contains :

- a list of contacts, with which connections can be made with the outer world.
- a list of parameters, through which each instance of a leafcell can have its own behaviour
- a leafcell body, that describes the behaviour

A compound contains :

- a list of contacts, with which connections can be made with the outer world.
- a list of parameters, which can be passed to the underlying compounds and leafcells
- a list of subsystems (instances of compounds and leafcells) used in this compound
- a list of wires, through which connections can be made in the compound itself
- a compound body, that
  - contains the calls to underlying compounds and leafcells
  - defines the interconnection between these subsystems, the wires and the contacts of both the subsystems and the compound itself

Note that only leafcells contain behaviour descriptions, and that only compounds contain information about the interconnection of the system and subsystems.

NDML does not know any predefined elements as SPICE does. However a library of commonly used leafcells is being constructed.

3. THE COMPILER

A compiler translates a text written in one language to a text with exactly the same meaning in another language.
In our case the object language is not machine code language, but a language of the same level. Such a compiler is by some people called a translator.
A compiler can be divided in a lexical analyser, a parser and a semantic part.

The lexical analyser scans the input to combine the input characters to logical units of the source language, such as for instance identifiers, numbers and keywords. These logical units are called lexical tokens. Lexical tokens are the input for the parser.

The parser checks if the input is written according to the syntax of SPICE input as described in appendix 2.

So called semantic actions are embedded in the parser. These actions perform the actual translation. They include writing data from the input into a data structure, operations on this data structure and writing from this data structure to the output file.

The next three chapters describe our implementation of these three parts of the compiler.

More about the theory of compilers and its parts can be found in [6], [7] and [8].

4. LEXICAL ANALYSIS

4.1 The Lexical Analyser Generator Lex

The lexical analyser has been made using the lexical analyser generator Lex [9]. The input for Lex is a list of regular expressions of input characters.

Backhouse, Aho and Ullman define a regular expression over the alphabet T (i.e. the set of characters that are used in a language) and the language denoted by that expression recursively as follows (See [6] from page 63 on, [7] from page 85 on) :

1. Ø is a regular expression denoting the empty set.
2. We use @ to denote the empty string.
   @ is a regular expression denoting {@}, that is the language containing only the empty string.
3. x , where x is an element of alphabet T, is a regular expression denoting {x}, i.e. the language containing only the string x.
4. If P and Q are regular expressions denoting the languages LP and LQ respectively, then
   - P|Q is a regular expression denoting the union of LP and LQ.
   - PQ is a regular expression denoting LP.LQ , i.e. all strings, which can be divided in two parts, while the first part is in LP and the second part is in LQ.
   - P* is a regular expression denoting the union of {@} , LP , LP.LP , LP.LP.LP and so on.

The alphabet used in Lex consists of all printable and control characters. Besides the regular expressions listed above Lex also recognises the following expressions :

1. [xy] the character x or y
2. [x-y] the characters x or y or any character between them in the alphabet
3. [^x] any character but x
4. . any character but newline
5. ^x an x at the beginning of a line
6. <S>x an x when Lex is in start condition S (see section 4.3)
7. x$ an x at the end of a line
8. x? an optional x
9. x+ one or more instances of x
10. x/y an x but only if followed by y
11. {c} an instance of the macro denoted by c (macros are defined in the so called definition section at the beginning of Lex input)
12. x{m,n} m through n occurrences of x

Note that the expressions 5, 6, 7 and 10 are context sensitive.

The regular expression action for recognising identifiers for instance is :

```
[a-zA-Z0-9][a-zA-Z0-9]*
```

Actions are added to each of the regular expressions in the input for Lex. When the input for the analyser, in our case SPICE input, matches one of these expressions, the corresponding actions are executed. Usually these actions

include returning a so called token to the parser, to indicate what kind of input has been found, for instance a number or an identifier.

4.2 SPICE Input

We have made regular expressions to recognise any SPICE input. An extra regular expression has been added for error detection (See chapter seven).

SPICE input consists of a set of so called cards. Each SPICE card ends with a newline. When this is followed by a + at the beginning of the next line however, the card continues here. Therefore the lexical analyser skips a newline followed by a + , but returns the token EOC (end of card) to the parser, when it finds a newline which is not followed by a + .

The parts of SPICE input that are captured by the lexical analyser using a so called start condition are discussed in the next section.

4.3 Start Conditions

A regular expression may be preceded by one or more so called start conditions. This expression can in that case only be matched, if one of the specified start conditions has been set. A start condition can be set anywhere both in the lexical analyser and in the parser. We have used this feature of Lex in order to make the parser less complicated, as will be explained below.

The first card in SPICE input is a so called title card. The start condition TITLEBEGIN has been used to return the title card in one token, called TITLE_EOC, to the parser. When we would not have used this start condition, the parser had to be extended to accept any token until the first newline had been found. In our case one simple grammar rule suffices to accept the token TITLE_EOC.

Next we have made regular expressions to recognise the beginning of each possible SPICE-card. Thereby the parser knows, from the first token after the token EOC, what kind of card is currently being parsed. All the regular expressions for recognising the beginning of a card are preceded by the start condition CARDBEGIN. This start condition is set after finding an end of a card. The SPICE capacitor card, for example, begins with the capital letter C . When this letter is found in the input at the beginning of a card, the token CLIT is returned to the parser.

When a name, e.g. a letter followed by any combination of letters, digits or underscores, is expected in the input, the start condition NAMEBEGIN is set. The name CLR for instance will be returned with the token NAME. If we would not have used start conditions, this name would be returned by the sequence of tokens for each character, in which case the parser should be extended with grammar rules to reduce this kind of sequences to the nonterminal name .

The start condition ANSTRINGBEGIN is set, whenever an alphanumeric string is expected. An alphanumeric string may begin with a digit or an underscore, while a name may not.

We use the start condition NOTBEGIN to catch keywords, that do not appear at the beginning of a card , and may not be returned with the token NAME.

When a number is found in the input, the start condition AFTERNUMBERBEGIN is set to skip any letters that SPICE allows just after a number. If there are no letters after a number, the start condition NOTBEGIN will be set.

Some cards in SPICE can not (yet) be translated to NDML. The start condition IGNOREBEGIN is set upon finding the beginning of such a card. The rest of the card, whatever it may be, will than be matched by a regular expression preceded by the start condition IGNOREBEGIN. The total card will then be skipped, e.g. no tokens will be returned to the parser.

The last card in the input must be an END card, beginning with the string .END . As soon as this string is found, the token END is returned to the parser and the rest of the input is skipped by the lexical analyser. Thus no error message will be given if the END card is not the last card of the input.

5. THE PARSER

The parser has been constructed using the parser generator Yacc [10]. Terms like grammar rules and context-free grammar are explained in appendix 1.

5.1 Usage of Yacc

The input for Yacc is a list of grammar rules with actions to be performed when the concerning rule is applied.

Yacc can produce two output files:

- a file containing the C source of the parser constructed
- a file containing a description of the parser constructed, in the form of a list with the states of the parser and the actions to be done when a certain lexical token is encountered

To the parser a lexical analyser must be added, that reads the input and divides it into lexical tokens. This analyser may be constructed by Lex. The parser calls the lexical analyser each time it needs the next token.

Yacc can produce parsers for almost every language which can be defined by a context-free grammar of class LALR(1).

5.2 Implementation

The grammar rules needed for Yacc can directly be derived from the syntax definition of SPICE input (see appendix 3). Only small modifications were needed to avoid reduce/reduce conflicts and shift/reduce conflicts.

The parser calls the lexical analyser constructed by Lex each time it needs the next token.

The way the parser handles errors and tries to recover after an error has been found is described in chapter seven.

6. SEMANTICS

The semantic actions that have been added to the parser perform the actual translation from SPICE input as described in chapter one to NDML as described in chapter two. These actions are put in functions that are called by the parser. Because Yacc generates a parser written in C, the semantic functions have been written in C too.

The semantic actions can be divided in the following classes :

- translating circuit elements
- translating subcircuits
- translating comments

In section 6.1 we will discuss translating a circuit element. In this section the need of buffers is shown. Section 6.2 deals with the use of buffers.

The translation of subcircuits and comments is described in the sections 6.3 and 6.4 .

The translation of some SPICE cards is not yet implemented. In section 6.5 a list is given of the cards that are not yet translated.

In the following sections some additional aspects of translating are described, such as the administration which leafcells are used in NDML, the composition of the right contact list and wire list in NDML, and the translation of numbers.

An example of SPICE input and the corresponding translated NDML text is given in figure 8.1 and 8.2 (See chapter 8).

6.1 Translating a Circuit Element

A circuit element, such as a resistor, a capacitor ,a semiconductor et cetera, is in SPICE instantiated in one

card, which at the same time is a call to one of the built-in predefined models of SPICE. In NDML each element type used must first be defined in a leafcell, e.g. a resistor with the resistance as parameter, a capacitor with the capacitance and initial voltage as parameters et cetera.

When an circuit element is found in SPICE input the following actions are executed :

- a leafcell definition is written to the beginning of the NDML output file
- to this leafcell is referred in the subsystem declaration part
- the node numbers in SPICE input are converted to contacts by preceding the numbers by the string n_ , and these contacts are added to the wire section (In section 6.6 we will see that the latter will not be done, if the contacts are already in the so called formal contact list)
- the element is placed in the compound body section

An example is given in figure 6.1 .

SPICE INPUT: NDML:

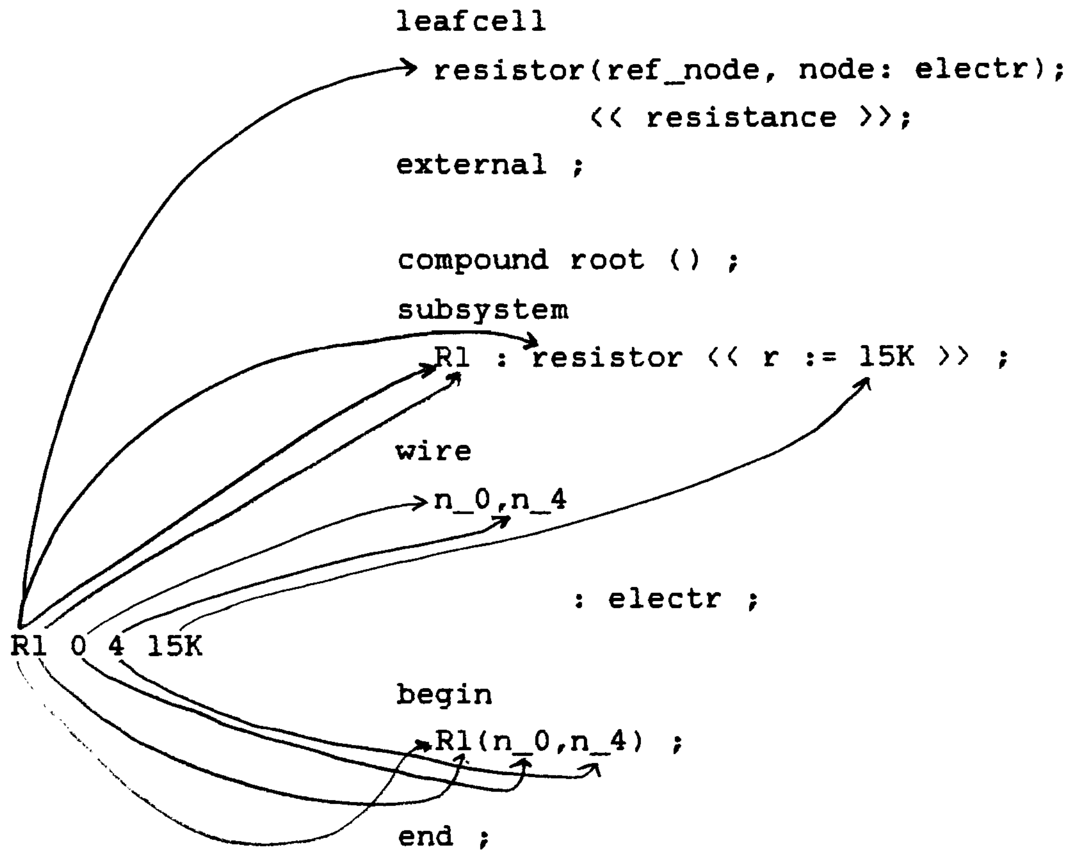

Figure 6.1 Translation of a circuit element

6.2 **Buffers Used for Temporary Storage**

From the example given in figure 6.1 we see that information from one SPICE card has to be stored in different parts of the NDML output. Therefore we have defined a set of five buffers to store the data for these parts.

These five buffers contain the text for :

- the compound heading
- the comments
- the subsystem declarations
- the wire section
- the compound body

When all circuit elements have been translated, these buffers will be flushed in this order into the output file.

6.3 The Translation of Subcircuits

In SPICE subcircuits can be defined. Subcircuits in SPICE may be nested. The definition and calls of these subcircuits may appear in either order. NDML also supports nesting of subcircuits. In NDML however a subcircuit (here called a compound) must be defined before it is called. Therefore the translated SPICE text related with another level of subcircuit nesting is stored in another set of buffers. The buffer set of a deeper nested subcircuit will be flushed before the buffer set of a lower nested one. Hereby each subcircuit in the NDML output will be defined before being called.

A subcircuit definition in SPICE begins with a subcircuit definition card and ends with an ENDS card. Each time a subcircuit definition card is found, the text is directed to a new (deeper nested) set buffers.
An ENDS card may contain a subcircuit name. The definition of the subcircuit with that name and the definitions of all deeper nested subcircuit are then terminated. When no name is given, all subcircuits definitions terminate. Upon finding an ENDS card the buffers sets of the subcircuit definitions being terminated are flushed into the output file in the right order. So if no name is given, all buffer sets but the buffer set for the main circuit are flushed.

Two subcircuits may have the same name provided that they are not defined within the same (sub)circuit. The compiler should translate subcircuits that have the same name to compounds with different names. This is not implemented yet. In that case the output of our compiler will contain two compounds with the same name, which is not allowed in NDML. The piecewise linear input compiler PLCOM will detect this error.

When an END card is found, a check is made to see whether or not all subcircuit buffer sets have been flushed. If not, an error message is generated. Next all buffers are flushed.

6.4 The Translation of Comment Cards

Any comment found in SPICE is surrounded by NDML comment marks (i.e. braces) and put in the comment buffer.
Braces in SPICE comment will be translated to brackets.
When flushing the buffers, the comment buffer is flushed after the compound heading buffer, but before the subsystem declaration buffer.

6.5 Cards That Are Not translated

The coupled inductor card and the transmission line card are the only element cards that are not translated yet.

The SPICE control cards that specify certain parameters of the models used and that specify what kind of output is wanted are also not translated.

These cards are:

.TEMP	.WIDTH	.OPTIONS	.OP	.DC
.NODESET	.IC	.TF	.SENS	.AC
.DISTO	.NOISE	.TRAN	.FOUR	.PRINT
.PLOT				

Warnings will be given both on the screen and in the list

file when such a card is found.

6.6 **Leafcells Used**

Information about which leafcells are used is stored during parsing. The definitions of the leafcells must appear in the output before the compounds. After the total input has been parsed, the definitions of the leafcells that have been used are copied from a library to the beginning of the output file.

All leafcells are defined external, so standard leafcells can be used, as well as user defined ones. In either case these real leafcell definitions have to be linked by the user with the output of this compiler.

6.7 **The Contact List and the Wire List**

In SPICE the connections between circuit elements are called nodes and denoted by an integer number. In NDML however connections inside a compound are made through so called wires. Connections of a compound with the outer world are made through so called contacts. Both contacts and wires are denoted by names. Therefore we let the name of a contact or wire in the output consist of the number of the corresponding SPICE node, preceded by the string n_ .

The main circuit has no contacts, all the nodes in the main circuit will be put in the wire list.

The nodes mentioned in the SPICE subcircuit definition card are the only contacts of that subcircuit with the outer world. Therefore all nodes in the definition that do not occur in the subcircuit definition card are converted in NDML to wires.

When a subcircuit definition card is found in the input, the nodes found in this card are put in the formal contact list of the compound corresponding with this subcircuit. Upon finding a node in the following cards in the input, the

presence of this node in the formal contact list is being checked. If this node does not occur there, it will be put in the wire list. Besides the node is put in the actual contact name list of the element concerned in the subsystem declaration section.

In a subcircuit definition in SPICE node 0 (ground) is always global, and may not appear in the subcircuit definition card's node list. Therefore we make in the output node 0 the first element in every compound contact list, except for the main compound.

6.8 The Translation of Numbers

A number in SPICE can be followed by a scale factor and arbitrary letters for comments. NDML knows the same scale factors, but not T (Tera) and MIL (milli inch). Milli is in SPICE M, and in NDML ML. Mega is in SPICE MEG and in NDML M. Care is taken of the right translation of the scale factors. If there are no digits before the decimal point, a 0 is placed there, because NDML does not accept numbers starting with a decimal point.

7. ERROR HANDLING

We expect the user to have run SPICE with the input file, before he translates this input file with our compiler. Therefore we may assume that a SPICE 2G input file, that is to be processed by the compiler, does not contain any errors.

Nonetheless our compiler can detect some errors and try to recover as explained in the next sections.

7.1 Errors Detected By The Lexical Analyser

We have made regular expressions to capture all legal SPICE input symbols. Also we have added a regular expression for characters that are not captured by one of the preceding legal SPICE input regular expressions.

This means that when this last regular expression is matched, an unexpected character is found in the input. A message will then be printed in the debug file, the list file and on the screen. When an unexpected character is found, nothing is returned to the parser and the lexical analyser proceeds with the next input character(s). In effect the unexpected character is skipped.

7.2 Errors Detected During Parsing

When the parser, as constructed by Yacc, discovers a syntax error the user redefinable function yyerror is called. Our version of yyerror causes an error message to be printed both on the screen and in the list file. The parser will then act as if it had received the token error , and will pop its stack until it enters a state where the token error is legal. Therefore we included a grammar rule

```
card : error EOC ;
```

in the Yacc input. Hereby the parser recovers from a syntax error by simply skipping the input until the first EOC (end

of card) is encountered.

Next we have included checks in the semantic functions, to verify that all subcircuit definitions are terminated by an ENDS card. If there are too many or too less ENDS cards an error message will be printed. Superfluous ENDS cards will be skipped. If there are not enough ENDS cards, they will be inserted just before the END card.

8. USAGE

The compiler is started by executing the file spicendml. It takes it input from the standard input. The compiler produces three files. These file are:

- an output file called output, which contains the translated SPICE input, i.e. NDML text
- a list file called list.tmp , that contains the SPICE input with error messages and warnings if any
- a debug file called debug.tmp , which indicates what tokens have been found by the lexical analyser, and what cards have been processed by the parser

All leafcells in the NDML output file are defined to be external. Hereby the NDML output can be compiled by the piecewise linear simulator input compiler PLCOM without the real leafcell definitions. The user has to link the PLCOM output of the real leafcell definitions with the PLCOM output of the NDML output file constructed by the SPICE input to NDML compiler. These real definitions may be both standard leafcells from a library and user defined ones.

The compiler does not translate SPICE transmission lines, coupled inductors and cards that specify certain parameters and what kind of output is wanted (See section 6.5). Warnings will be given both on the screen and in the list file when such a card is found.

Some subcircuits in SPICE may have the same name. Our compiler does not translate these kinds of subcircuits correctly.
See section 6.3 .

An example of the SPICE input to NDML translation is given below. Figure 8.1 gives the SPICE input, while figure 8.2 contains the NDML output of the compiler.

```
ORIGINAL FILE B7700/(ELESIC20)PADDRIVER ON USER19

VDD 1 0 5V
CL 7 0 8PF
X1  0 1 SUBC1

.SUBCKT SUBC1 1 2
C33 1 2      22PF
X12 1 2      SUBC2

.SUBCKT SUBC2 1 2
VIN 3 0 DC 0V
R11 1 2      2K
C21 2 7      10PF
M1  1 2 2 7  MODD L=6U W=6U AD=78P
M2  2 3 0 8  MODE L=6U PD=12U PS=130U
.ENDS SUBC1

.PRINT I(V1) V(1)

.MODEL MODD NMOS(LEVEL=2 VTO=-3.97)
.MODEL MODE NMOS(LEVEL=2 VTO=0.68 GAMMA=0.28)

.END
```

Figure 8.1 An example of SPICE input

```
leafcell MODD(drain, gate, source, bulk : electr) ;
        { NMOS ; LEVEL=2 ; VTO=-3.97 }
        << L ; W ; AD ; AS ; PD ; PS ; NRD ; NRS ; OFF ;
           ICVDS ; ICVGS ; ICVBS
        >> ;
external ;

leafcell MODE(drain, gate, source, bulk : electr) ;
        { NMOS ; LEVEL=2 ; VTO=0.68 ; GAMMA=0.28 }
        << L ; W ; AD ; AS ; PD ; PS ; NRD ; NRS ; OFF ;
           ICVDS ; ICVGS ; ICVBS
        >> ;
external ;

leafcell v_source (plus,minus : electr); << E >> ;
external ; {voltage source}

leafcell resistor(ref_node, node : electr);
        << resistance >>;
external ; { electrical resistor }

leafcell capacitor(ref_node, node : electr);
        << capacitance;
           V_init : default 0;
        >>;
external ; { electrical capacitor }

{ title in SPICE was :
ORIGINAL FILE B7700/(ELESIC20)PADDRIVER ON USER19
}
```

Figure 8.2.a The NDML output of the compiler.

See next page for figure 8.2.b .

```
compound SUBC2 (n_0,n_1,n_2  : electr) ;
subsystem
  VIN : v_source << E:=0 >> ;
  R11 : resistor << r:=2K >> ;
  C21 : capacitor << c:=10P >> ;
  M1 : MODD << L:=6U , W:=6U , AD:=78P >> ;
  M2 : MODE << L:=6U , PD:=12U , PS:=130U >> ;
wire  n_3,n_7,n_8 : electr ;
begin
  VIN(n_3,n_0) ;
  R11(n_1,n_2) ;
  C21(n_2,n_7) ;
  M1(n_1,n_2,n_2,n_7) ;
  M2(n_2,n_3,n_0,n_8) ;
end ;

compound SUBC1 (n_0,n_1,n_2  : electr) ;
subsystem
  C33 : capacitor << c:=22P >> ;
  X12 : SUBC2 ;
begin
  C33(n_1,n_2) ;
  X12(n_0,n_1,n_2) ;
end ;

compound root () ;
subsystem
  VDD : v_source << E:=5 >> ;
  CL : capacitor << c:=8P >> ;
  X1 : SUBC1 ;
wire  n_0,n_1,n_7 : electr ;
begin
  VDD(n_1,n_0) ;
  CL(n_7,n_0) ;
  X1(n_0,n_0,n_1) ;
end ;
```

Figure 8.2.b The NDML output of the compiler.

LITERATURE

[1] van Bokhoven, W.M.G.
Piecewise Linear Modelling and Analysis
Eindhoven, Netherlands: Research Group ES, Eindhoven University of Technology, 1981, Ph.D. Thesis.

[2] van Eijndhoven, J.T.J.
A Piecewise Linear Simulator for Large Integrated Circuits
Eindhoven, Netherlands: Research Group ES, Eindhoven University of Technology, 1984, Ph.D. Thesis.

[3] Vladimirescu, A et al.
SPICE Version 2G User's Guide
Berkeley, California: Department of Electrical Engineering and Computer Science, University of California, 1981

[4] Verschueren, A.C.
Storage Structures for the Eindhoven University of Technology Network Description and Modelling Compiler
Eindhoven, Netherlands: Research Group ES, Eindhoven University of Technology, 1984.

[5] Janssen, G.L.J.M.
Handleiding voor het gebruik van het Simulatiepakket (In Dutch), 1st ed.
Eindhoven, Netherlands: Research Group ES, Eindhoven University of Technology, 1984, Internal Report

[6] Backhouse, R.C.
Syntax of Programming Languages
London: Prentice-Hall, 1979

[7] Aho, A.V. and J.D. Ullman
Principles of Compiler Design
Reading, Massachusetts: Addison-Wesley, 1977

[8] Bauer, F.L. et al.
Compiler Construction, An Advanced Course, 2nd ed.
Berlin: Springer

[9] Lesk, M.E. and E. Schmidt
Lex - A Lexical Analyzer Generator
Murray Hill, New Jersey: Bell Laboratories, 1975.
Computer Science Technical Report No. 39.

[10] Johnson, S.C.
Yacc: Yet Another Compiler-Compiler
Murray Hill, New Jersey: Bell Laboratories, 1978.
Computer Science Technical Report No. 32.

APPENDIX 1: CONTEXT-FREE GRAMMARS AND BNF

The syntax of a language can be given in the form of a context-free grammar or in the extended Backus-Naur form. We will give here a definition of both.

A context-free grammar is defined by the quadruple (N,T,S,P), where (See [6], pages 7 and 13) :

1. N is a finite set of non-terminal symbols
2. T is a finite set of terminal symbols
3. S is an element of N, called the start symbol
4. P is a set of productions (also called grammar rules) each of which have the form A --> v where A is a non-terminal and v is an arbitrary combination of non-terminals and terminals, even zero non-terminals and zero terminals is allowed.

The language defined by this grammar is the set of all strings that can be derived with the productions from the start symbol.

The extended Backus-Naur form EBNF also uses the following special characters in the right part of a production:

- | to denote choice
- the curly braces { and } to denote zero or more instances of what is between them
- the brackets [and] to denote zero or one instance of what is between them
- the parentheses (and) to denote higher precedence for what is between them

Instead of the right arrow --> EBNF uses ::= in a production.

Non-terminals in EBNF are surrounded by < and > .

Terminals are surrounded by double quotes " .

A production is terminated with a period . .

We can now give a recursive definition of the extended Backus-Naur form. Note that everything between (* and *) is comment.

```
<syntax>         ::= { <production-rule> } .

<production-rule> ::=
           <nonterminal-symbol> "::=" <expression> "." .

<expression>     ::= <term> { "|" <term> } .

(* The | denotes choice. The expression can be derived to
   one of the terms listed.
*)

<term>           ::= <factor> { <factor> } .

<factor>         ::=
           <nonterminal-symbol>
         | <terminal-symbol>
         | "(" <expression> ")"
                (* The parentheses denote higher precedence
                   for <expression> *)
         | "{" <expression> "}"
                (* The curly braces denote zero or more
                   instances of <expression> *)
         | "[" <expression> "]" .
                (* The brackets denote zero or one instance
                   of <expression> *)
```

The language defined by this grammar is the set of all strings that can be derived with the productions from the start symbol.

A terminal is a character or sequence of characters surrounded by double quotes. Every string between less than and greater than characters < en > is a non-terminal. We use pseudo non-terminals for terminals that we can not

describe clearly with the double quotes notation.
<TAB> is a pseudo-non-terminal symbol denoting a tabulation.
<EOL> is a pseudo-non-terminal symbol denoting an end of line.

APPENDIX 2: THE SYNTAX OF SPICE 2G INPUT

The general form of each card as described in SPICE Version 2 G User's Guide precedes the head production rule of each card.

We assume that newline + sequences in the input have been skipped, i.e. a card, followed by card(s) with + in the first column, or card(s) starting with separators and + , is seen now as one card. (See [3], page 6).

There is now no limitation on the length of cards.

In the following sections everything between (* and *) is comment.

1. Higher Part

```
<input-deck>    ::= <title-card> {<card>} <end-card> .

<card>          ::=
           [<separators>] ( <comment-card>
                          | <element-card>
                          | <period-card>
                          | <subckt-call-card>) .

<period-card>  ::=
           <model-card>   |
           <subckt-card>  | <ends-card>    | <temp-card>  |
           <width-card>   | <options-card> | <op-card>    |
           <dc-card>      | <nodeset-card> | <ic-card>    |
           <tf-card>      | <sens-card>    | <ac-card>    |
           <disto-card>   | <noise-card>   | <tran-card>  |
           <four-card>    | <print-card>   | <plot-card>  .
```

```
<element-card> ::=
          <resistor>               | <capacitor>            |
          <inductor>               | <coupled-inductor>     |
          <transmission-line>      | <lin-vccs>             |
          <lin-vcvs>               | <lin-cccs>             |
          <lin-ccvs>               | <independent-vs>       |
          <independent-cs>         | <junction-diode>       |
          <bjt>                    | <jfet>                 |
          <mosfet>                 .
```

2. Subcircuit-calls

See [3] page 8.

```
(* XYYYYY N1 [N2 N3 ...] SUBNAM *)      ( See [3] page 29 )

<subckt-call-card> ::=
          <XLIT> <alphanumeric-string>
          <separators>   <node>
          <separators>   [<next-node> <separators>]
                         <name-field>
          [<separators>] <EOL> .

<next-node>     ::= <node> <separators> [<next-node>] .
```

3. Cards Beginning With a Period

See [3] page 18-41. We use here the non-terminal any-symbol-not-EOL because we are not interested in the actual contents, since we, for the time being, will not translate these contents to NMDL.

```
<model-card>   ::= ".MODEL" {<any-symbol-not-EOL>} <EOL> .

(* .SUBCKT subnam N1 [N2 N3 N4 ...] *)
<subckt-card>  ::=
          ".SUBCKT"
          <separators> <name-field> <separators> <node>
          [<separators> [<next-node-opt-separators>]]
          <EOL> .
```

```
<next-node-opt-separators> ::=
          <node>
          [<separators> [<next-node-opt-separators>]] .

<ends-card>      ::=
          ".ENDS"
          [<separators>] [[<name>] <separators>] <EOL> .

<temp-card>      ::= ".TEMP" {<any-symbol-not-EOL>} <EOL> .

<width-card>     ::= ".WIDTH" {<any-symbol-not-EOL>} <EOL> .

<options-card> ::=
          ".OPTIONS" {<any-symbol-not-EOL>} <EOL> .

<op-card>        ::=
          ".OP"
          [<separators> {<any-symbol-not-EOL>}] <EOL> .

<dc-card>        ::= ".DC" {<any-symbol-not-EOL>} <EOL> .

<nodeset-card> ::=
          ".NODESET" {<any-symbol-not-EOL>} <EOL> .

<ic-card>        ::= ".IC" {<any-symbol-not-EOL>} <EOL> .

<tf-card>        ::= ".TF" {<any-symbol-not-EOL>} <EOL> .

<sens-card>      ::= ".SENS" {<any-symbol-not-EOL>} <EOL> .

<ac-card>        ::= ".AC" {<any-symbol-not-EOL>} <EOL> .

<disto-card>     ::= ".DISTO" {<any-symbol-not-EOL>} <EOL> .

<noise-card>     ::= ".NOISE" {<any-symbol-not-EOL>} <EOL> .

<tran-card>      ::= ".TRAN" {<any-symbol-not-EOL>} <EOL> .

<print-card>     ::= ".PRINT" {<any-symbol-not-EOL>} <EOL> .

<plot-card>      ::= ".PLOT" {<any-symbol-not-EOL>} <EOL> .
```

4. Resistor, Capacitor, (Coupled) Inductor

See [3] page 9-11.

```
(* RXXXXX N1 N2 VALUE [TC=TC1 [,TC2]] *)
<resistor>      ::=
            <RLIT> <alphanumeric-string>
            <separators> <node>
            <separators> <node>
            <separators> <number-field>
            [<separators> [<TC-part>]] <EOL> .

<TC-part>       ::=
            <TCLIT> <equal-sign-opt-separators>
            <number-field> [<comma-part>] .

(* NOTE : TC = )= 0.4 () , .21 =) is allowed in SPICE 2G *)

<comma-part>    ::=
            {<separators-not-comma>}
            ["," [<separators>]
             [<number-field> [<separators>]]] .

(* CXXXXX N+ N- VALUE [IC=INCOND] *)
(* CXXXXX N+ N- POLY L0 L1 L2 ... [IC=INCOND] *)
<capacitor>     ::= <CLIT> <capacitor-inductor-common-part> .

(* LXXXXX N+ N- VALUE [IC=INCOND] *)
(* LXXXXX N+ N- POLY L0 L1 L2 ... [IC=INCOND] *)
<inductor>      ::= <LLIT> <capacitor-inductor-common-part> .

<capacitor-inductor-common-part> ::=
            <alphanumeric-string>
            <separators> <node>
            <separators> <node>
            <separators>
            (<number-field> [<separators> [<IC-part>]]
            | <POLYLIT> <separators> <number-field>
                            <to-the-right-of-number-field> )
            <EOL> .

<IC-part>       ::=
            <ICLIT> <equal-sign-opt-separators>
            <number-field> [<separators>] .
```

```
<to-the-right-of-number-field> ::=
          <EOL>
          | <separators> ( <EOL>
                         | <another-number-field>
                         | <IC-part> <EOL>        ) .

<another-number-field> ::=
          <number-field> <to-the-right-of-number-field> .

(* KXXXXX LYYYYY LZZZZZ VALUE *)
<coupled-inductor> ::=
          <KLIT> <alphanumeric-string>
          <separators> <LLIT> <alphanumeric-string>
          <separators> <LLIT> <alphanumeric-string>
          <separators> <number-field> [<separators>]
          <EOL> .
```

5. Transmission-line

See [3] page 11.

```
(* TXXXXX N1 N2 N3 N4 Z0=VALUE
       [TD=VALUE] [F=FREQ [NL=NRMLEN]] [IC=V1,I1,V2,I2]  *)
<transmission-line> ::=
          <TLIT> <alphanumeric-string>
          <separators>  <node>
          <separators>  <node>
          <separators>  <node>
          <separators>  <node>
          <separators>  <Z0LIT> <equal-sign-opt-separators>
                        <number-field>
          [<separators> [<after-Z0>]] <EOL> .

<after-Z0>      ::=
          <TD> [<separators> [<after-TD>]]
        | <after-TD> .
```

```
<after-TD>      ::=
           <F> [<separators> ([<NL>
                                [<separators> [<after-F>]]]
                              | <after-F>                  )]
         | <after-F> .

<after-F>       ::= <IC> {<separator>} .

<TD>            ::=
           <TDLIT> <equal-sign-opt-separators> <VALUE> .

<F>             ::=
           <FLIT> <equal-sign-opt-separators> <VALUE> .

<NL>            ::=
           <NLLIT> <equal-sign-opt-separators> <VALUE> .

<IC>            ::=
           <ICLIT> <equal-sign-opt-separators>
           <VALUE> "," <VALUE> "," <VALUE> "," <VALUE> .
```

6. Controlled Sources

See [3] page 12-17,47-49.

```
(* GXXXXX N+ N- NC+ NC- VALUE *)
(* GXXXXX N+ N- <POLY(ND)>
          NC1+ NC1- ... P0 <P1 ...> <IC=...> *)
<lin-vccs>      ::= <GLIT> <volt-cntrld-sources-common-part>.

(* EXXXXX N+ N- NC+ NC- VALUE *)
(* EXXXXX N+ N- <POLY(ND)>
          NC1+ NC1- ... P0 <P1 ...> <IC=...> *)
<lin-vcvs>      ::= <ELIT> <volt-cntrld-sources-common-part>.
```

```
<volt-cntrld-sources-common-part> ::=
          <alphanumeric-string>
          <separators> <node>
          <separators> <node>
          <separators> <node>
          <separators> <node>
          <separators> <number-field>
          [<separators>] <EOL> .

(* FXXXXX N+ N- VNAM VALUE *)
(* FXXXXX N+ N- <POLY(ND)>
         VN1 <VN2 ...> P0 <P1 ...> <IC=...> *)
<lin-cccs>     ::= <FLIT> <curr-cntrld-sources-common-part>.

(* HXXXXX N+ N- VNAM VALUE *)
(* HXXXXX N+ N- <POLY(ND)>
         VN1 <VN2 ...> P0 <P1 ...> <IC=...> *)
<lin-ccvs>     ::= <HLIT> <curr-cntrld-sources-common-part>.

<curr-cntrld-sources-common-part> ::=
          <alphanumeric-string>
          <separators> <node>
          <separators> <node>
          <separators> <alphanumeric-string>
          <separators> <number-field>
          [<separators>] <EOL> .
```

7. Independent Sources

See [3] page 14-17.

```
(* VXXXXX N+ N- [[DC] DC/TRAN VALUE]
                [AC  [ACMAG  [ACPHASE]]]   [<after-AC>]   *)
<independent-vs>   ::=  <VLIT>  <independent-sources-common-
part>.

(* IXXXXX N+ N- [[DC] DC/TRAN VALUE]
                [AC  [ACMAG  [ACPHASE]]]   [<after-AC>]   *)
<independent-cs>   ::=  <ILIT>  <independent-sources-common-
part>.
```

```
<independent-sources-common-part> ::=
           <alphanumeric-string>
           <separators> <node>
           <separators> <node>
           [<separators> [<independent-sources-last-part>]]
           <EOL> .

<independent-sources-last-part> ::=
           <DC> [<separators> [<after-DC>]]
         | <after-DC> .

<DC>            ::= [<DCLIT> <separators>] <number-field> .

<after-DC>      ::=
           <ACLIT> [<separators>
                 ([<number-field>
                   [<separators>  ([<number-field>
                                    [<separators>
                                     [<after-AC>]]]
                                  | <after-AC>      ) ]]
                 | <after-AC>                            )]
         | <after-AC> .

<after-AC>      ::=
           (* PULSE(V1 V2 TD TR TF PW PER) *)
           <PULSELIT> <lpar-opt-separators>
                          <number-field>
             <separators> <number-field>
             <separators> <number-field>
             <separators> <number-field>
             <separators> <number-field>
             <separators> <number-field>
             <separators> <number-field>
             <rpar-opt-separators>
            (* SIN(V0 VA FREQ TD THETA) *)
         | <SINLIT> {<separator-not-lpar>} "("
             {<separator>} <number-field>
             <separators> <number-field>
             <separators> <number-field>
```

```
              <separators> <number-field>
              <separators> <number-field>
              <rpar-opt-separators>
            (* EXP(V1 V2 TD1 TAU1 TD2 TAU2) *)
          | <EXPLIT> <lpar-opt-separators>
                          <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <rpar-opt-separators>
           (* PWL(T1 V1 [T2 V2 T3 T4 ...]) *)
         | <PWLLIT> <lpar-opt-separators>
                          <number-field>
              <separators> <number-field>
              <pwl-right-recursion>
            (* SFFM(V0 VA FC MDI FS) *)
          | <SFFMLIT> {<separator-not-lpar>} "("
              {<separator>} <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <separators> <number-field>
              <rpar-opt-separators> .

<pwl-right-recursion> ::=
            ")" [<separators>] [<extra-Ti-Vi-pair>]
          | <separator-not-rpar> {<separator-not-rpar>}
              ( ")" [<separators>] [<extra-Ti-Vi-pair>]
              | <extra-Ti-Vi-pair>                      ) .

<extra-Ti-Vi-pair> ::=
            <number-field> <separators>
            <number-field> <pwl-right-recursion> .
```

8. Semiconductors

See [3] page 18-21.

```
(* DXXXXX N+ N- MNAME [AREA] [OFF] [IC=VD] *)
<junction-diode> ::=
           <DLIT> <alphanumeric-string>
           <separators> <node>
           <separators> <node>
           <separators> <name-field>
           [<separators> [<diode-after-string>]] <EOL> .

<diode-after-string> ::=
           <number-field>
             [<separator> [<diode-after-area>]]
         | <diode-after-area> .

<diode-after-area> ::=
           <OFFLIT> [<separators> [<diode-after-off>]]
         | <diode-after-off> .

<diode-after-off> ::=
           <ICLIT> <equal-sign-opt-separators>
           <number-field> {<separator>} .

(* QXXXXX NC NB NE [NS] MNAME [AREA] [OFF] [IC=VBE,VCE] *)
<bjt>         ::=
           <QLIT> <alphanumeric-string>
           <separators> <node>
           <separators> <node>
           <separators> <node>
           <separators> [<node> <separators>]
                        <name-field>
           [<separators> [<bjt-after-string>]]
           <EOL> .
```

```
<bjt-after-string> ::=
          <number-field> [<separator> [<bjt-after-area>]]
        | <bjt-after-area> .

<bjt-after-area> ::=
          <OFFLIT> [<separators> [<bjt-after-off>]]
        | <bjt-after-off> .

<bjt-after-off> ::=
          <ICLIT> <equal-sign-opt-separators>
           <number-field> <comma-opt-separators>
           <number-field> {<separator>} .


(* JXXXXX ND NG NS MNAME [AREA] [OFF] [IC=VDS,VGS] *)
<jfet>          ::=
          <JLIT> <alphanumeric-string>
          <separators> <node>
          <separators> <node>
          <separators> <node>
          <separators> <name-field>
          [<separators> [<jfet-after-string>]]
          <EOL> .

<jfet-after-string> ::=
          <number-field> [<separator> [<jfet-after-area>]]
        | <jfet-after-area> .

<jfet-after-area> ::=
          <OFFLIT> [<separators> [<jfet-after-off>]]
        | <jfet-after-off> .


<jfet-after-off>::=
          <ICLIT> <equal-sign-opt-separators>
          <number-field> <comma-opt-separators>
          <number-field> {<separator>}.
```

```
(* MXXXXX ND NG NS NB MNAME [L=VAL] [W=VAL] [AD=VAL]
          [AS=VAL] [PD=VAL] [PS=VAL] [NRD=VAL] [NRS=VAL]
          [OFF] [IC=VDS,VGS,VBS]                            *)
<mosfet>        ::=
           <JLIT> <alphanumeric-string>
           <separators> <node>
           <separators> <node>
           <separators> <node>
           <separators> <node>
           <separators> <name-field>
           [<separators> [<mosfet-after-string>]]
           <EOL> .

<mosfet-after-string> ::=
           <LLIT> <equal-sign-opt-separators> <number-field>
           [<separators> [<mosfet-after-l>]]
         | <mosfet-after-l> .

<mosfet-after-l> ::=
           <WLIT> <equal-sign-opt-separators> <number-field>
           [<separators> [<mosfet-after-w>]]
         | <mosfet-after-w> .

<mosfet-after-w> ::=
           <ADLIT><equal-sign-opt-separators> <number-field>
             [<separators> [<mosfet-after-ad>]]
         | <mosfet-after-ad> .

<mosfet-after-ad> ::=
           <ASLIT><equal-sign-opt-separators> <number-field>
             [<separators> [<mosfet-after-as>]]
         | <mosfet-after-as> .

<mosfet-after-as> ::=
           <PDLIT><equal-sign-opt-separators> <number-field>
             [<separators> [<mosfet-after-pd>]]
         | <mosfet-after-pd> .
```

```
<mosfet-after-pd> ::=
            <PSLIT><equal-sign-opt-separators> <number-field>
              [<separators> [<mosfet-after-ps>]]
          | <mosfet-after-ps> .

<mosfet-after-ps> ::=
            <NRDLIT><equal-sign-opt-separators><number-field>
              [<separators> [<mosfet-after-nrd>]]
          | <mosfet-after-nrd> .

<mosfet-after-nrd> ::=
            <NRSLIT><equal-sign-opt-separators><number-field>
              [<separators> [<mosfet-after-nrs>]]
          | <mosfet-after-nrs> .

<mosfet-after-nrs> ::=
            <OFFLIT> [<separators> [<mosfet-after-off>]]
          | <mosfet-after-off> .

<mosfet-after-off> ::=
            <ICLIT> <equal-sign-opt-separators>
            <number-field> <comma-opt-separators>
            <number-field> <comma-opt-separators>
            <number-field> {<separator>}.
```

9. Lexical Rules

See [3] page 6-8.

```
<title-card>   ::= {<any-symbol-not-EOL>} <EOL> .

<comment-card> ::= "*" {<any-symbol-not-EOL>} <EOL> .

<end-card>     ::= ".END" {<any-symbol-not-EOL>} <EOL> .

<name-field>   ::= <letter> {<alphanumeric-element>} .

<node>         ::= <digits> .
```

(* NOTE :

Spice allows the separators to be placed between any pair of adjacent fields.
Therefore we define the following 4 non-terminals, to denote that, if one of the separators has to occur as a field (as in TC = TC1, see the resistor card), this separator has to occur at least once between to two neighbour fields (TC and TC1 in this case).
TC =(= TC1 is allowed !
*)

```
<equal-sign-opt-separators> ::=
          {<separator-not-equal-sign>} "=" {<separator>}.

<comma-opt-separators> ::=
          {<separator-not-comma>} "," {<separator>} .

<lpar-opt-separators> ::=
          {<separator-not-lpar>} "(" {<separator>} .

<rpar-opt-separators> ::=
          {<separator-not-rpar>} ")" {<separator>} .

<separators>    ::= <separator> {<separator>} .

<separator>     ::= <separator-not-comma> | "," .

<separator-not-comma> ::= " " | "=" | "(" | ")" .

<separator-not-lpar>  ::= " " | "=" | ")" | "," .

<separator-not-rpar>  ::= " " | "=" | "(" | "," .

<separator-not-equal-sign>  ::= " " | "(" | ")" | "," .

<number-field> ::= [<sign>] <unsigned-number-field> .

<unsigned-number-field> ::=
          <digits> ["." [<digits>]] [<number-field-part>]
        | "." <digits> [<number-field-part>] .
```

```
<number-field-part> ::=
            <integer-exponent> [<letters>]
          | <scale-factor-not-M> [<letters>]
          | "M" [<eg-il-letters>]
          | <letter-not-EFGKMNPTU> [<letters>] .

<integer-exponent> ::= "E" [<sign>] <digits> .

<scale-factor-not-M> ::= "T"|"G"|"K"|"U"|"N"|"P"|"F" .

<eg-il-letters> ::=
            "E" [<letters-after-ME>]
          | "I" [<letters-after-MI>]
          | <letter-not-EI> [<letters>] .

<letters-after-ME> ::=
            "G" [<letters>]
          | <letter-not-G> [<letters>] .

<letters-after-MI> ::=
            "L" [<letters>]
          | <letter-not-L> [<letters>] .

<any-symbol-not-EOL> ::=
            <letter>
          | <digit>
          | <TAB>
          | <special-character> .

<sign>          ::= "+" | "-" .

<digits>        ::= <digit> {<digit>} .

<alphanumeric-string> ::=
            <alphanumeric-element> {<alphanumeric-element>}.

<alphanumeric-element> ::=
            <letter>
          | <digit>
          | <spice-special-character> .

<letters>       ::= <letter> {<letter>} .
```

```
<special-character> ::=
          <spice-special-character>
        | "["
        | " "
        | ","
        | "("
        | ")"
        | "=" .

(* separators are " " | "," | "(" | ")" | "="  *)

(* SPICE 2G on the Burroughs 7700 fails to create an output
   file if it finds a left bracket [ in the input file. *)

<spice-special-character> ::=
          "!" | """" | "#" | "$" | "%" | "&" | "'" | "*"
        | "+" | "-"  | "." | "/" | ":" | ";" | "<" | ">"
        | "?" | "@"  | "
        | "|" | "}"  | "~" .

<period>                ::= "." .
<left-parenthesis>      ::= "(" .
<right-parenthesis>     ::= ")" .
<left-curly-brace>      ::= "{" .
<right-curly-brace>     ::= "}" .
<left-bracket>          ::= "[" .
<right-bracket>         ::= "]" .

(* lower-case letters are not allowed in SPICE input *)

<letter>       ::= <letter-not-G> | "G" .

<letter-not-G> ::= <letter-not-EGIL> | "E" | "I" | "L" .

<letter-not-L> ::= <letter-not-EGIL> | "E" | "G" | "I" .

<letter-not-EI> ::= <letter-not-EGIL> | "G" | "L" .

<letter-not-EGIL> ::=
          <letter-not-EFGIKLMNPTU>
        | "F" | "K" | "M" | "N" | "P" | "T" | "U" .
```

```
<letter-not-EFGKMNPTU> ::=
          <letter-not-EFGIKLMNPTU> | "I" | "L" .

<letter-not-EFGIKLMNPTU> ::=
          "A" | "B" | "C" | "D" | "H" | "J" | "O" | "Q"
        | "R" | "S" | "V" | "W" | "X" | "Y" | "Z" .

<digit>        ::=
          "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
        | "8" | "9" .

<ACLIT>        ::= "AC" .
<ADLIT>        ::= "AD"
<ASLIT>        ::= "AS"
<CLIT>         ::= "C"
<DLIT>         ::= "D"
<DCLIT>        ::= "DC"
<ELIT>         ::= "E" .
<EXPLIT>       ::= "EXP" .
<FLIT>         ::= "F" .
<GLIT>         ::= "G" .
<HLIT>         ::= "H" .
<ILIT>         ::= "I" .
<ICLIT>        ::= "IC" .
<JLIT>         ::= "J" .
<KLIT>         ::= "K" .
<LLIT>         ::= "L" .
<MLIT>         ::= "M" .
<NLLIT>        ::= "NL" .
<NRDLIT>       ::= "NRD" .
<NRSLIT>       ::= "NRS" .
<OFFLIT>       ::= "OFF" .
<PDLIT>        ::= "PD" .
<POLYLIT>      ::= "POLY" .
<PSLIT>        ::= "PS" .
<PULSELIT>     ::= "PULSE" .
<PWLLIT>       ::= "PWL" .
<QLIT>         ::= "Q" .
```

<RLIT> ::= "R" .
<SINLIT> ::= "SIN" .
<SFFMLIT> ::= "SFFM" .
<TLIT> ::= "T" .
<TCLIT> ::= "TC" .
<TDLIT> ::= "TD" .
<VLIT> ::= "V" .
<WLIT> ::= "W" .
<XLIT> ::= "X" .
<Z0LIT> ::= "Z0" .

APPENDIX 3: DESCRIPTION OF FUNCTIONS, VARIABLES AND MACROS USED

This appendix contains a short functional description of the most important semantic functions, variables and macros used in the compiler.
An alphabetic list of the names of these items and the page numbers where they are described is included at the end of this appendix.

All semantic functions called by the parser have a name that starts with sem . When the title card has been found, the function **semTITLE** is called. When a comment card has been found, the functions **semCOMM10** and **semCOMM999** are called, et cetera.

1. Functions Invoked by Main

The main function of the compiler is called **main.** Main initialises some arrays and file pointers. Next it calls the function generated by YACC **yyparse** (see chapter .) to parse the input. After parsing, the definitions of the leafcells used are copied into the totaloutputfile named output and summaries for errors and warnings are printed, if there are any, on the screen. Hereafter the outputfile of the parser named output.tmp is appended to the totaloutputfile. Three more files are produced by main, which will be discussed below.

1.1 Listing File

The compiler produces a list file with name list.tmp . This file contains a listing of the input read, with line numbers and error messages. The input is echoed to the listfile by the lexical analyser through the predefined macros **ECHOA** and **ECHOO.** These predefined macros invoke the function **copylist,** to echo the input found by the lexical analyser to the listing file. **ECHOO** (ECHO Only) echoes the input only to the listing file. **ECHOA** (ECHO Also) also copies the input

to the debug file. **Copylist** has an character array as its argument. It copies this array to the list file. When it encounters a newline character, the following actions are executed :

- the predefined macro **CHECKLINENRONPAGE** is called, which checks if there is still space for a next line on the current page. Otherwise it will send a form feed to the list file.
- the predefined macro **NEWLINE** is called, which sends a newline character to the list file, and calls the function **copyerror** (see error handling and recovery) if an error message or warning has been issued in the line, terminated by this newline. This is indicated by the as boolean used integer **errorinthisline.**

After parsing, **main** calls the functions **errorsummary** and **endlist.** The first function will print an error summary on the screen and in the list file, the latter will print the number of lines processed.

1.2 Debug File

Extra information about the actions of the lexical analyser and the parser is stored in the debug file with name debug.tmp . The lexical analyser prints in this file a message each time a regular expressions is matched by a part of the input. The parser prints which cards are found. Besides this file contains all error messages and warnings.

1.3 Statistics File

This file contains information about the space used by the buffers. These figures can be compared with the maximum buffer length, which is also given in this file.

2. General Functions

2.1 The Translation Of Subcircuits

The nesting level of subcircuits is during parsing stored in the extern variable **actbufnr** (actual buffer number). **actbufnr** determines in which buffer level text is put. Initially this variable has got the value 0, denoting the main circuit.

The function **copystring** stores a string in one of the buffers. This function has three parameters. The first parameter must be the nesting level and determines in which buffer level the string will be stored. The second denotes in which buffer the string is stored : the compound header, the comment section, the subsystem declaration part, the wire section or the compound body. To denote these buffers we have defined the macros **COMP, COMM, SUBS, WIRE** and **BODY.**

A subcircuit definition in SPICE begins with a subcircuit definition card and ends with an ENDS card. Each time a subcircuit definition card is found, **actbufnr** will be incremented by one, and the name of the subcircuit is stored in the character pointer array **subcktname.**

An ENDS card may contain a subcircuit name. This name indicates which subcircuit definition is terminated. When no name is given, all subcircuits definitions terminate. Upon finding an ENDS card **actbufnr** is decremented until the name in the array **subcktname** at index **actbufnr** corresponds with the given name. If no name is given, **actbufnr** is decremented to 0. In either case, for each decrement by one of **actbufnr,** the corresponding set of buffers is flushed into the output file using the function **fprintfbuf.** This function has the number of the buffer to be flushed as its argument. When an END card is found, the function **flushbuf** is called. This function checks if all subcircuit definitions have been closed, by inspecting the variable **actbufnr.** If not, an error message is generated. All

buffers are flushed, using the function **fprintfbuf**.

2.2 The Translation Of Comment Cards

Any comment found in SPICE is surrounded by NDML comment marks (braces) and put in the comment buffer. Braces is SPICE comment are converted to brackets. The function **commenttobuf** takes care of these actions.

2.3 The Translation Of Numbers

A number in SPICE can be followed by a scalefactor and arbitrary letters for comments. NDML knows the same scalefactors, but not T (Tera) and MIL (milli inch). Milli is in SPICE M, and in NDML ML. Mega is in SPICE MEG and in NDML M. The function **copyvalue** takes care of the right translation of the scalefactors. If there are no digits before the decimal point, it places a "0" there, because NDML does not accept numbers starting with a decimal point.

2.4 Leafcells Used

Information about which leafcells are used is stored in the as boolean array used integer array **leafcellused**. Therefore the macro **TRUE** is defined to be 1, while **FALSE** is defined to be 0. Each leafcell is assigned an integer number by a C define statement. If a certain leafcell is used, the element in the array **leafcellused** at the index that corresponds with the number assigned to the leafcell concerned is set to **TRUE**. The definitions of the leafcells must appear before the compounds. After the total input has been parsed, the array **leafcellused** is checked by the function **collectleafcells**. For each leafcell that has been used, e.g. whose value in the array **leafcellused** is **TRUE**, the corresponding file, which contains the definition of this leafcell, is copied to the totaloutputfile. Thereafter the the outputfile is appended to the totaloutputfile.

2.5 The Contact List and The Wire List

In SPICE the connections between circuit elements are called nodes and denoted by an integer number. In NDML however connections inside a compound are made through so called wires. Connections of a compound with the outer world are made through so called contacts.

We store the node numbers found in the input in the three dimensional integer array **node.** The first dimension of this array denotes in which subcircuit nesting level we have found the node concerned. The second dimension denotes if the node is an internal one or an external one, e.g. in NDML terms if it is a contact or a wire. Therefore we have defined the constants **INT** and **EXT** to be used as indices for this dimension. An two dimensional integer array **nodeffi** is used to store the indices of the first free element (first free indices) in the third dimension of the array **node.**

The main circuit has no contacts. The nodes mentioned in the subcircuit definition card are the only contacts of that subcircuit. All nodes in the definition that do not occur in the subcircuit definition card are converted in NDML to wires. Therefore we have the functions **nodein** and **addnode.** The function **nodein** returns an as boolean used integer, and has 3 parameters. These 3 parameters are respectively the subcircuit nesting level, the index for the second dimension of the array node, and a node number. The function returns **TRUE** if the node number is in the subarray of **node** corresponding with the given nesting level and given index for the second dimension. The function **addnode** has the same parameters. It adds the node number to the subarray concerned and returns in that case **TRUE.** If the node number already exists in this subarray it returns the value **FALSE.**

When a subcircuit definition card is found in the input, the nodes found in this card are put in the array **node** using the function **addnode.** Upon finding a node in other cards in the input, the function **processnode** is called. This function puts the name of the node in the actual contact name list of

the element concerned in the buffer for the compound body. Besides it calls the function **nodein** to check if the node is already in the formal contact list of the compound being constructed, or in other words, if the node occurred in the subcircuit definition card in SPICE input. If not, the function **addnode** is called to add this node in the wire list. The name of a contact or wire in the output consists of the number of the corresponding SPICE node, preceded by the string "n_".

In a subcircuit definition in SPICE node 0 (ground) is always global, and may not appear in the subcircuit definition card's node list. Therefore we make in the output node 0 the first element in every compound contact list (see function **semSUB30**), except for the main compound.

3. Error Handling

A two dimensional array called **errorlist** has been created for recording the errors made in each input line. At the end of each input line, a message is printed in the debug file and on the screen, when errors have been found. In the first column of the array errorlist the column number where the error occurred is saved. The second column contains the type number of the error. This array can be filled using the function **fillerrorlist.** This function needs one parameter, namely the type number of the error. When the array **errorlist** is not full, this type number is stored in the first free place in the second column, while the global variable **colnr,** which is the column number in the list file, is stored on the same row in the first column of this array. To be able to print the whole line on the screen, each line is temporarily stored in the character array **lastline.**

When an error occurred the function **copyerror** will be called at the end of the current line either by the macro **NEWLINE,** which is called when a newline character has been send to the list file, or by the function **contonnextline,** which is executed when the number of characters send to the list file

exceeds the maximum column number **MAXCOL.** The function **copyerror** displays the current line on the screen using the character array **lastline** and prints both on the screen and in the list file below the current line the place and the type number of the error(s) found, using the array **errorlist.** The function **errorsummary,** which is called by the main program **main,** will print an error summary on the screen and at the end of the list file.

3.1 Errors Detected by the Parser

When the parser, constructed by Yacc, discovers a syntax error the user redinable function **yyerror** is called. In our case this function calls the function **fillerrorlist.** Also the parser will act as if it had received the token error , and will pop its stack until it enters a state where the token error is legal. Therefore we included a grammar rule

```
card : error EOC ;
```

in the Yacc input. Hereby the parser can recover by skipping the input until the first EOC (end of card), when it finds a syntax error. When finding this first EOC the statements **yyerrok** and BEGIN CARDBEGIN, and the function **aftererror** are executed. The statement **yyerrok** resets the parser to its normal state. Otherwise a syntax error at the beginning of the next line might not be reported by the parser. The statement BEGIN CARDBEGIN enables Lex to find the right token at the beginning of the next card. The function **aftererror** calls the function **undo** which undoes the text in the buffers related with the current card, by placing comment marks. Also the function **aftererror** resets the as boolean used integer variable **to_be_undone** to **FALSE.**

3.2 Errors Detected by the Lexical Analyser

Errors detected by the lexical analyser are handled in the following way. A message will be printed in the debug file. If in the current line did not occur a syntax error until now and if there are more than 5 spare places for errors in

the array **errorlist**, than also a message will be printed on the screen and in the listfile. The last 5 places in the array **errorlist** are reserved for more important errors.

4. Alphabetic List

name	page
actbufnr	56
addnode	58,59
aftererror	59,60
BODY	56
CHECKLINENRONPAGE	55
collectleafcells	57
colnr	59
COMM	56
commenttobuf	57
COMP	56
contonnextline	59
copyerror	55,59,60
copylist	54,55
copystring	56
copyvalue	57
ECHOA	54
ECHOO	54
endlist	55
errorinthisline	55
errorlist	59,60,61
errorsummary	55,60
EXT	58
FALSE	57
fillerrorlist	59,60
flushbuf	56
fprintfbuf	56,57
INT	58
lastline	59,60
leafcellused	57
main	54,55,60
MAXCOL	60
NEWLINE	55,59
node	58
nodeffi	58
nodein	58,59
processnode	58
semCOMM10	54
semCOMM999	54
semSUB30	59
semTITLE	54
subcktname	56
SUBS	56
to_be_undone	60
TRUE	57
undo	60
WIRE	56
yyerror	60
yyerrok	60
yyparse	54