# Parsing Spice Netlists Using a Typed Functional Language

Nicolas Ratier, Mahmoud Addouche, Daniel Gillet, Rémi Brendel

## ▶ To cite this version:

## HAL Id: hal-00345200
### https://hal.science/hal-00345200

Submitted on 8 Dec 2008

# Parsing Spice Netlists Using a Typed Functional Language

N. RATIER, M. ADDOUCHE, D. GILLET, R. BRENDEL
Laboratoire de Physique et Métrologie des Oscillateurs
associé à l'Université de Franche–Comté
32 avenue de l'Observatoire, 25044 Besançon Cedex
FRANCE
nicolas.ratier@lpmo.edu     http://www.lpmo.edu/~ratier

*Abstract:* - Parsing a Spice netlist is the first step of all circuit simulation programs. This part is usually done by low-level coding techniques in *C* or *Fortran* language. The aim of this paper is to show the usefulness of functional programming techniques to the needs of scientific computing.

*Key-Words:* - Spice Netlist, Parsing, Scanning, Functional Language.

## 1 Introduction

Despite a reasonably long history, functional techniques are seldom used in numerical programs. This is mainly because they remain relatively unknown to the scientific computing community. However, each time a problem involves trees, pattern-matching, or is naturally expressed as recursive definitions, functional languages are perfectly suitable.

In this paper we consider the problem of parsing a Spice [3] netlist using a functional language. We focus on the problem of transforming a netlist to an intermediate form which can be easily turned into any other syntax, for example the Matlab or Maple syntaxes.

We choose the *Gentle* [4] typed functional language because it can be easily linked with C programs, transformed into a `dll`, or compiled to make a stand-alone executable. We assume that the reader is acquainted with scanning and parsing techniques.

## 2 Program structure

The textual form of a Spice netlist is not adapted to transformation treatments. Code transformation is thus done in two successive steps:

- The first step transforms the netlist into abstract syntax trees. The use of trees reflects the netlist structures (subcircuit nesting, model description, ...). The building of the abstract syntax tree from the netlist involves the usual simultaneous tasks:

  1. **Scanning**: The lexical analyzer recognizes Spice keywords and transforms the input text into a sequence of tokens.
  2. **Parsing**: A parser recognizes the syntax and performs the semantic actions attached to relevant grammars rules. They are responsible for building the abstract syntax tree.

- The second step generates code from a transformed abstract syntax tree. All manipulations are based on tree traversals and code transformation becomes tree transformation followed by code generation:

  1. **Semantic analysis**: Information about identifiers may not be deduced from the syntax only (for example, whether a subcircuit has been declared or not, ...). These informations are found during code transformation and recorded in a symbol table.
  2. **Code transformation**: Code transformation mainly flats subcircuit content. It assigns a unique name to subcircuit nodes that are not formal parameters and to device names in the nested subcircuits.
  3. **Code generation**: A valid Matlab code is generated from the transformed abstract syntax tree and the associated symbol table.

If we want to change the generated code into another syntax, say that of *Maple* or *Mathematica*, only the code generation part has to be changed. This code is essentially pretty-printing code and is very easy to write.

## 3 Implementation
### 3.1 Lexical analysis

We have used *flex* [2] to generate the scanner which recognizes lexical patterns in text. Certain Spice features and our aim to write a simple and well

structured grammar led us to enlarge the role of the lexical analyzer in three areas:

- treatment of the first line in the input file (in Spice this is the title line),

- handling of continuation lines (a line may be continued by entering a "+" in column 1 of the following line),

- parsing of components values (this handles scale factors to compute values such as 10kHz).

## 3.2 Abstract syntax

The so–called abstract syntax is perfectly suited as a basis for the specification of program transformations, and is used as an intermediate program representation. The abstract syntax is elaborated from the Spice concrete syntax, and represented as a set of 'type' in *Gentle*.

For example, a Spice netlist is a list of statements and control lines. A statement is a list of circuit components, model parameters or subcircuit definitions. It is expressed in the *Gentle* language by defining the type STATEMENT with five constituents (called *functors*): seq, component, model, subcircuit, nil. The complete definitions of the 'type' NETLIST and STATEMENT are:

```
'type' NETLIST
   seq(NETLIST,NETLIST)
   statement(STATEMENT)
   controlline(CONTROLLINE)
   nil

'type' STATEMENT
   seq(STATEMENT,STATEMENT)
   component(COMPONENT)
   model(STRING,STRING,MODELPARAM)
   subcircuit(STRING,
              SUBCIRCUITNODE,
              STATEMENT)
   nil

'type' COMPONENT
   resistor(STRING,STRING,...)
   ...
   mesfet(STRING,STRING,STRING,...)
   callsubcircuit(STRING,
                  SUBCIRCUITNODE,
                  STRING)
```

The fifth line of the type STATEMENT means that a subcircuit is composed of its name, the list of subcircuit nodes, and its subcircuit content.

- We see here that the type STATEMENT is recursive (a STATEMENT can contain a STATEMENT), allowing to express naturally that a subcircuit can be nested.

- Subcircuit definitions may contain anything (other subcircuit definitions, device models, ...) but control lines. This is the reason why the functor seq appears both in the NETLIST and STATEMENT types.

## 3.3 Translating concrete syntax into abstract syntax

We now show how concrete syntax can be translated into abstract syntax. Concrete syntax is described by grammar rules in machine-readable BNF (Backus-Naur Form). It can parse exactly the same class of language that the *bison* (or *yacc*) parser can handle [1]. In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead.

Consider the program fragment in Fig. 1. When the source program is translated into an internal representation, rules are selected by the parser according to the given source. Each nonterminal gets an output parameter that specifies the corresponding abstract syntax. Terminal symbols appear in quotes (i.e. ".end") or are introduced by token declarations (like RESISTOR) and fully defined in the *flex* file. The tree, representing the abstract syntax, is progressively built at each reduce step of the parser.

## 3.4 Symbol table

Our analyzer attaches a symbol table to the syntax tree of a Spice netlist. This table holds the necessary informations about identifiers (nodes list, subcircuit and model names) used in the netlist. For example, it manages efficiently the following features of a Spice program:

- any device models or subcircuit definitions included as part of a subcircuit definition are strictly local,

- any element nodes not included on the .SUBCKT line are strictly local, with the exception of 0 (ground) which is always global.

A second and important property is the detection of semantic errors such as definition of a subcircuit already declared. The symbol table is mainly updated during the subcircuits calls.

We use the strategy described in [4](ch 6.3.8) to associate a meaning to each identifier. With this

```
'nonterm' Netlist(-> NETLIST)
   'rule' Netlist(-> A) : DeclarationList(-> A) ".end"

'nonterm' DeclarationList(-> NETLIST)
   'rule' DeclarationList(-> seq(B,nil)) :
           Declaration(-> B)
   'rule' DeclarationList(-> seq(B,A))   :
           DeclarationList(-> A) Declaration(-> B)

'nonterm' Declaration(-> NETLIST)
   'rule' Declaration(-> statement(A))   : Statement  (-> A)
   'rule' Declaration(-> controlline(A)) : ControlLine(-> A)

'nonterm' Statement(-> STATEMENT)
   'rule' Statement(-> component(A)) : ElementStatement      (-> A)
   'rule' Statement(-> component(A)) : SourceStatement       (-> A)
   'rule' Statement(-> component(A)) : SemiconductorStatement(-> A)
   'rule' ...

'nonterm' ElementStatement(-> COMPONENT)
   'rule' ElementStatement(-> resistor (N,A,B,C)) :
           RESISTOR (-> N) Node(-> A) Node(-> B) Value(-> C)
   'rule' ElementStatement(-> inductor (N,A,B,C)) :
           INDUCTOR (-> N) Node(-> A) Node(-> B) Value(-> C)
   'rule' ElementStatement(-> capacitor(N,A,B,C)) :
           CAPACITOR(-> N) Node(-> A) Node(-> B) Value(-> C)

'token' RESISTOR(-> STRING)
```

Fig. 1: Program fragment to translate concrete syntax into abstract syntax.

strategy, accessing the meaning of identifiers has the same cost as in the case of a flat name space. It is done in constant time because the symbol table is implemented as a hash table.

### 3.5   Code transformation

We have just described how a source program is translated into an internal representation (abstract syntax tree). We now discuss how the internal representation is translated into a target code. The mapping is specified in exactly the same way as within the parser, with the exception that abstract syntax now becomes an input parameter that controls rule selection. It can be viewed as the opposite work of parsing.

A fragment of the code transformation part is given in Fig. 2. The code generation for resistance is called by `A_Resistor` (with the necessary parameters) which is basically pretty-printing code.

The subcircuit call `A_Subcircuitcall` statement is more difficult to process. For example, its treatment must be aware that the order of lines

is arbitrary: a subcircuit can be called before it is defined in the netlist.

## 4   Efficiency

We can make a quantitative estimation of our program by counting the number of code lines excluding whitelines and comments. The following table illustrates the efficiency of functional programming in term of compactness.

| Abstract syntax | 44 | lines |
|---|---|---|
| Concrete syntax | 110 | lines |
| Symbol table | 111 | lines |
| Code transformation | 145 | lines |

It is interesting to compare the same computation written in *Gentle* and in *C* language. The Spice3f4 source file `subckt.c` handles imbedded `".subckt"` definitions (replacing formal nodes, flat device and node names, ...) is $954$ lines long. The same task is managed by our program with only $261$ lines.

```
'action' A_Netlist(NETLIST)
  'rule' A_Netlist(seq(A,nil))    :                A_Netlist(A)
  'rule' A_Netlist(seq(A,B))      : A_Netlist(B) A_Netlist(A)
  'rule' A_Netlist(statement  (A)) : A_Statement(A)
  'rule' A_Netlist(controlline(A)) : A_Controlline(A)

'action' A_Statement(STATEMENT)
  'rule' A_Statement(seq(A,nil))   :                A_Statement(A)
  'rule' A_Statement(seq(A,B))     : A_Statement(B) A_Statement(A)
  'rule' A_Statement(component(A)) : A_Component(A)
  ...

'action' A_Component(COMPONENT)
  'rule' A_Component(resistor(N,A,B,C))     : A_Resistor(N,A,B,C)
  'rule' A_Component(inductor(N,A,B,C))     : A_Inductor(N,A,B,C)
  'rule' A_Component(capacitor(N,A,B,C))    : A_Capacitor(N,A,B,C)
  ...
  'rule' A_Component(mesfet(N,A,B,C,D))     : A_Mesfet(N,A,B,C,D)
  'rule' A_Component(subcircuitcall(N,A,B)) : A_Subcircuitcall(N,A,B)
```

Fig. 2: Program fragment to transform an abstract syntax tree.

# 5 Conclusion

We have tried to demonstrate the applicability of modern functional programming to the needs of scientific computing. A program to parse a Spice netlist and transform it into another syntax have been written with only 410 code lines. Moreover, to change the syntax of the generated code only a very straightforward part of the program has to be rewritten.

In the context of writing source analysis and transformation programs, the key to the efficiency of functional programming is that it provides a very natural representation of abstract syntax trees. Especially tree traversals and rewriting are well expressed with the pattern matching facility.

We have shown that functional programming provides clear, readable and semantically very powerful code. For many problems, it is much more productive than the C/C++ language. To summarize, it has the following advantages for all source transformation related tasks:

- less time and effort is required to write the program,

- the resulting implementation is easier to maintain and modify.

*References:*

[1] Charles Donnelly and Richard Stallman. *Bison, The YACC-compatible Parser Generator*, November 1995. version 1.25.

[2] Vern Paxson. *Flex, A fast scanner generator*, March 1995. version 2.5.

[3] T. Quarles, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli. *Spice 3 User Manual*. Department of Electrical Engineering and Computer Sciences, University of California, 1993. version 3f4.

[4] Friedrich Wilhelm Schröer. *The GENTLE Compiler Construction System*. R. Oldenbourg Verlag, 1997. ISBN 3-486-24703-4.