INFO-F403 - INTRODUCTION TO LANGUAGE THEORY AND COMPILING

PART 2: PARSER SUPER FORTRAN COMPILER

November 21, 2018

GEISSER AGUILAR NIKLAUS
ngeisser@ulb.ac.be

GERARD SALINAS JOAN S.
jgerards@ulb.ac.be

Contents

1 Grammar Transformation

In order to implement the parser, it was necessary to transform the given grammar to LL(1), to do so, the following algorithms have been applied:

1.1 Removal of useless symbols

No useless symbols were found in the grammar applying the algorithms of removal of unproductive and inaccessible symbols. We decided to not add the steps that we followed since nothing from the original grammar has changed up until this point.

1.2 Removal of ambiguity

The following rules of the grammar are ambiguous since two different derivation trees can be generated for the same input. Moreover, these rules do not respect priority and associativity of the given operators.

1.2.1 Arithmetic operators

This section of the grammar representing the arithmetic expressions generation was identified as ambiguous.

<ExprArith>	→	[VarName]
	→	[Number]
	→	(<ExprArith>)
	→	<ExprArith><op><ExprArith>
<op>	→	+
	→	−
	→	*
	→	/

This was transformed into the following:

<ExprArith>	→	<ExprArith>+<ExprMult>
	→	<ExprArith>-<ExprMult>
	→	<ExprMult>
<ExprMult>	→	<ExprMult>*<ID>
	→	<ExprMult>/<ID>
	→	<Id>
<Id>	→	[VarName]
	→	[Number]
	→	(<ExprArith>)
	→	− <IdTail>
<IdTail>	→	[VarName]
	→	[VarNumber]

By doing this transformation, this part of the grammar is not ambiguous anymore.
Note that the **<IdTail>** variable has been added in order to avoid the production of three or more consecutive minus symbols. e.g: "− − − <Id>"

1.2.2 Binary operators

The original grammar is presented below.

<Cond>	→	<Cond><BinOp><Cond>
	→	NOT <SimpleCond>
<SimpleCond>	→	<ExprArith> <Comp> <ExprArith>
<BinOp>	→	AND
	→	OR

It was transformed into the following:

<Cond>	→	<Cond>OR<CondAnd>
	→	<CondAnd>
<CondAnd>	→	<CondAnd>AND<CondFinal>
	→	<CondFinal>
<CondFinal>	→	NOT <Cond>
	→	<SimpleCond>

By doing this transformation, this part of the grammar is not ambiguous anymore.

1.3 Left Factorization

The left factorization algorithm was applied for three sections in the grammar.

1.3.1 If condition

The following part of the grammar can be left factorized:

<If>	→	IF (<Cond>) THEN [EndLine] <Code> ENDIF
	→	IF (<Cond>) THEN [EndLine] <Code> ELSE [EndLine] <Code> ENDIF

And transformed into the following:

<If>	→	IF (<Cond>) THEN [EndLine] <Code><IfTail>
<IfTail>	→	ENDIF
	→	ELSE [EndLine] <Code> ENDIF

1.3.2 Varname

The original grammar is presented below.

<VarList>	→	[VarName], <VarList>
	→	[VarName]

And transformed into the following:

<VarList>	→	[VarName] <VarListTail>
<VarListTail>	→	, <VarList>
	→	ϵ

Notice that the same transformation was applied for <ExpList> and <Id> rules.

1.4 Removal of left recursion

The following parts of the grammar present left recursion:

1.4.1 Arithmetic operations

The grammar after removal of ambiguity is presented below.

<ExprArith>	→	<ExprArith>+<ExprMult>
	→	<ExprArith>-<ExprMult>
	→	<ExprMult>
<ExprMult>	→	<ExprMult>*<ID>
	→	<ExprMult>/<ID>
	→	<Id>

Applying the Removal of left recursion algorithm, the result will be:

<ExprArith>	→	<ExprMult><ExprArithA>
<ExprArithA>	→	+ <ExprMult> <ExprArithA>
	→	− <ExprMult> <ExprArithA>
	→	ϵ
<ExprMult>	→	<Id><ExprMultA>
<ExprMultA>	→	$*$ <Id><ExprMultA>
	→	/ <Id><ExprMultA>
	→	ϵ

1.4.2 Binary operations

The grammar after removal of ambiguity is presented below.

<Cond>	→	<Cond><CondAnd>
	→	<CondAnd>
<CondAnd>	→	<CondAnd>AND<CondFinal>
	→	<CondFinal>

Applying the Removal of left recursion algorithm, the result will be:

<Cond>	→	<CondAnd><CondA>
<CondA>	→	OR <CondAnd><CondA>
	→	ϵ
<CondAnd>	→	<CondFinal> <CondAndA>
<CondAndA>	→	AND <CondFinal> <CondAndA>
	→	ϵ
<CondFinal>	→	NOT <SimpleCond>
	→	<SimpleCond>

As it can be seen, there is no left recursion in the grammar.

2 New Grammar

As a result of applying all the algorithms described above, the new grammar is:

1	<Program>	$\rightarrow$	BEGINPROG [ProgName] [EndLine] <Variables> <Code> ENDPROG [EOS]
2	<Variables>	$\rightarrow$	VARIABLES <VarList> [EndLine]
3		$\rightarrow$	ϵ
4	<VarList>	$\rightarrow$	[VarName] <VarListTail>
5	<VarListTail>	$\rightarrow$	, <VarList>
6		$\rightarrow$	ϵ
7	<Code>	$\rightarrow$	<Instruction> [EndLine] <Code>
8		$\rightarrow$	ϵ
9	<Instruction>	$\rightarrow$	<Assign>
10		$\rightarrow$	<If>
11		$\rightarrow$	<While>
12		$\rightarrow$	<For>
13		$\rightarrow$	<Print>
14		$\rightarrow$	<Read>
15	<Assign>	$\rightarrow$	[VarName] := <ExprArith>
16	<ExprArith>	$\rightarrow$	<ExprMult><ExprArithA>
17	<ExprArithA>	$\rightarrow$	+ <ExprMult> <ExprArithA>
18		$\rightarrow$	− <ExprMult> <ExprArithA>
19		$\rightarrow$	ϵ
20	<ExprMult>	$\rightarrow$	<Id><ExprMultA>
21	<ExprMultA>	$\rightarrow$	$*$ <Id><ExprMultA>
22		$\rightarrow$	/ <Id><ExprMultA>
23		$\rightarrow$	ϵ
24	<Id>	$\rightarrow$	[VarName]
25		$\rightarrow$	[Number]
26		$\rightarrow$	(<ExprArith>)
27		$\rightarrow$	− <IdTail>
28	<IdTail>	$\rightarrow$	[VarName]
29		$\rightarrow$	[VarNumber]

30	<If>	→	IF (<Cond>) THEN [EndLine] <Code><IfTail>
31	<IfTail>	→	ENDIF
32		→	ELSE [EndLine] <Code> ENDIF
33	<Cond>	→	<CondAnd><CondA>
34	<CondA>	→	OR <CondAnd><CondA>
35		→	ϵ
36	<CondAnd>	→	<CondFinal> <CondAndA>
37	<CondAndA>	→	AND <CondFinal> <CondAndA>
38		→	ϵ
39	<CondFinal>	→	NOT <SimpleCond>
40		→	<SimpleCond>
41	<SimpleCond>	→	<ExprArith><Comp><ExprArith>
42	<Comp>	→	=
43		→	>=
44		→	>
45		→	<=
46		→	<
47		→	<>
48	<While>	→	WHILE (<Cond>) DO [EndLine]<Code> ENDWHILE
49	<For>	→	FOR [VarName] := <ExprArith> TO <ExprArith> DO [EndLine] <Code> ENDFOR
50	<Print>	→	PRINT(<ExpList>)
51	<Read>	→	READ(<VarList>)
52	<ExpList>	→	<ExprArith> <ExpListTail>
53	<ExpListTail>	→	, <ExpList>
54		→	ϵ

3 FIRST(1)

<Program>	BEGINPROG
<Variables>	VARIABLES ϵ
<VarList>	[VarName]
<VarListTail>	, ϵ
<Code>	[VarName] IF WHILE FOR PRINT READ ϵ
<Instruction>	[VarName] IF WHILE FOR PRINT READ
<Assign>	[VarName]
<ExprArith>	[VarName] [Number] (–
<ExprArithA>	$+ - \epsilon$
<ExprMult>	[VarName] [Number] (–
<ExprMultA>	$* / \epsilon$
<Id>	[VarName] [Number] (–
<IdTail>	[VarName] [VarNumber]
<If>	IF
<IfTail>	ENDIF ELSE
<Cond>	NOT [VarName] [Number] (–
<CondA>	OR ϵ
<CondAnd>	NOT [VarName] [Number] (–
<CondAndA>	AND ϵ
<CondFinal>	NOT [VarName] [Number] (–
<SimpleCond>	[VarName] [Number] (–
<Comp>	= >= > <= < <>
<While>	WHILE
<For>	FOR
<Print>	PRINT
<Read>	READ
<ExpList>	[VarName] [Number] (–
<ExpListTail>	, ϵ

4 FOLLOW(1)

<Program>	
<Variables>	[VarName] IF WHILE FOR PRINT READ ENDPROG
<VarList>	[EndLine]
<VarListTail>	[EndLine]
<Code>	ENDPROG ENDIF ESE ENDWHILE ENDFOR
<Instruction>	[EndLine]
<Assign>	[EndLine]
<ExprArith>	[EndLine] ,) = >= > <= < <> TO DO AND OR
<ExprArithA>	[EndLine] ,) = >= > <= < <> TO DO AND OR
<ExprMult>	+ −[EndLine] ,) = >= > <= < <> TO DO AND OR
<ExprMultA>	+ −[EndLine] ,) = >= > <= < <> TO DO AND OR
<Id>	∗ / + −[EndLine] ,) = >= > <= < <> TO DO AND OR
<IdTail>	∗ / + −[EndLine] ,) = >= > <= < <> TO DO AND OR
<If>	[EndLine]
<IfTail>	[EndLine]
<Cond>	)
<CondA>	)
<CondAnd>	OR)
<CondAndA>	OR)
<CondFinal>	AND OR)
<SimpleCond>	AND OR)
<Comp>	[VarName] [Number] (−
<While>	[Endline]
<For>	[Endline]
<Print>	[Endline]
<Read>	[Endline]
<ExpList>	)
<ExpListTail>	)

5 ACTION TABLE

The action table is not presented as a table since there is a limit of space when displaying. That's why you can find the detailed action table at the folder "more" of the project. Note that when a terminal symbol is read from the input it will be matched.

- <Program>
  - Rule (1) -> BEGINPROG
- <Variables>
  - Rule (2) -> VARIABLES.
  - Rule (3) -> ENDPROG [VarName] IF WHILE FOR PRINT READ.
- <VarList>
  - Rule (4) -> [VarName]
- <VarListTail>
  - Rule (5) -> ,
  - Rule (6) -> [EndLine]
- <Code>
  - Rule (7) -> [VarName] IF WHILE FOR PRINT READ
  - Rule (8) -> ENDPROG ENDIF ELSE ENDWHILE ENDFOR
- <Instruction>
  - Rule (9) -> [VarName]
  - Rule (10) -> IF
  - Rule (11) -> WHILE
  - Rule (12) -> FOR
  - Rule (13) -> PRINT
  - Rule (14) -> READ
- <Assign>
  - Rule (15) -> [VarName]
- <ExprArith>

  - Rule (16) -> – [VarName] [Number] (

- <ExprArithA>
  - Rule (17) -> +
  - Rule (18) -> –
  - Rule (19) -> [EndLine]) OR AND = >= > <= < <> DO TO ,
- <ExprMult>
  - Rule (20) -> – [VarName] [Number] (
- <ExprMultA>
  - Rule (21) -> ∗
  - Rule (22) -> /
  - Rule (23) -> [EndLine] + –) OR AND = >= > <= < <> DO TO ,
- <Id>
  - Rule (24) -> [VarName]
  - Rule (25) -> [Number]
  - Rule (26) -> (
  - Rule (27) -> –
- <IdTail>
  - Rule (28) -> [VarName]
  - Rule (29) -> [Number]
- <If>
  - Rule (30) -> IF
- <IfTail>
  - Rule (31) -> ENDIF
  - Rule (32) -> ELSE
- <Cond>
  - Rule (33) -> – [VarName] [Number] (NOT

- <CondA>
  - Rule (34) -> OR
  - Rule (35) ->)
- <CondAnd>
  - Rule (36) -> – [VarName] [Number] (NOT
- <CondAndA>
  - Rule (37) -> AND
  - Rule (38) ->) OR
- <CondFinal>
  - Rule (39) -> NOT
  - Rule (40) -> – [VarName] [Number] (
- <SimpleCond>
  - Rule (41) -> – [VarName] [Number] (
- <Comp>
  - Rule (42) -> =
  - Rule (43) -> >=
  - Rule (44) -> >
  - Rule (45) -> <=
  - Rule (46) -> <
  - Rule (47) -> <>
- <While>
  - Rule (48) -> WHILE
- <For>
  - Rule (49) -> FOR
- <Print>
  - Rule (50) -> PRINT

- <Read>
  - Rule (51) -> READ
- <ExpList>
  - Rule (52) -> – [VarName] [Number] (
- <ExpListTail>
  - Rule (53) -> ,

6 Recursive descent LL(1) Parser

In order to continue the implementation of the parser, a recursive descent parser was written in Java, following the production rules described in the new grammar. The parser receives the tokens from the Lexical analyzer and starts the process with the first production rule. Whenever the parser detects an unexpected token, it stops the compilation process triggering a syntax error showing the line and column of the error along with the expected value. e.g:

```
SYNTAX ERROR near line 17, column 18.
Please add an end of line before {ENDIF}

PRINT(result) ENDIF
              ^
```

Note that the symbol ˆ was used to denote where exactly the syntax error occurs.
However, if the lexical analyzer already detected any error the parser process will be never executed.
If there are no syntax errors in the code provided, the parser will output the left most derivation along and internally it will construct the parse tree that will be represented by figure 1 which was generated for the following code:

```
BEGINPROG AllGrammar

    VARIABLES result, someothervar
ENDPROG
```

The numbers next to the lexical units in figure 1 represent the order in which the tokens have been matched. Each terminal symbol from the input is highlighted to illustrate the derivation.

Regarding the [EndLine] terminal symbol, only the first one found will be matched and all others will be skipped.

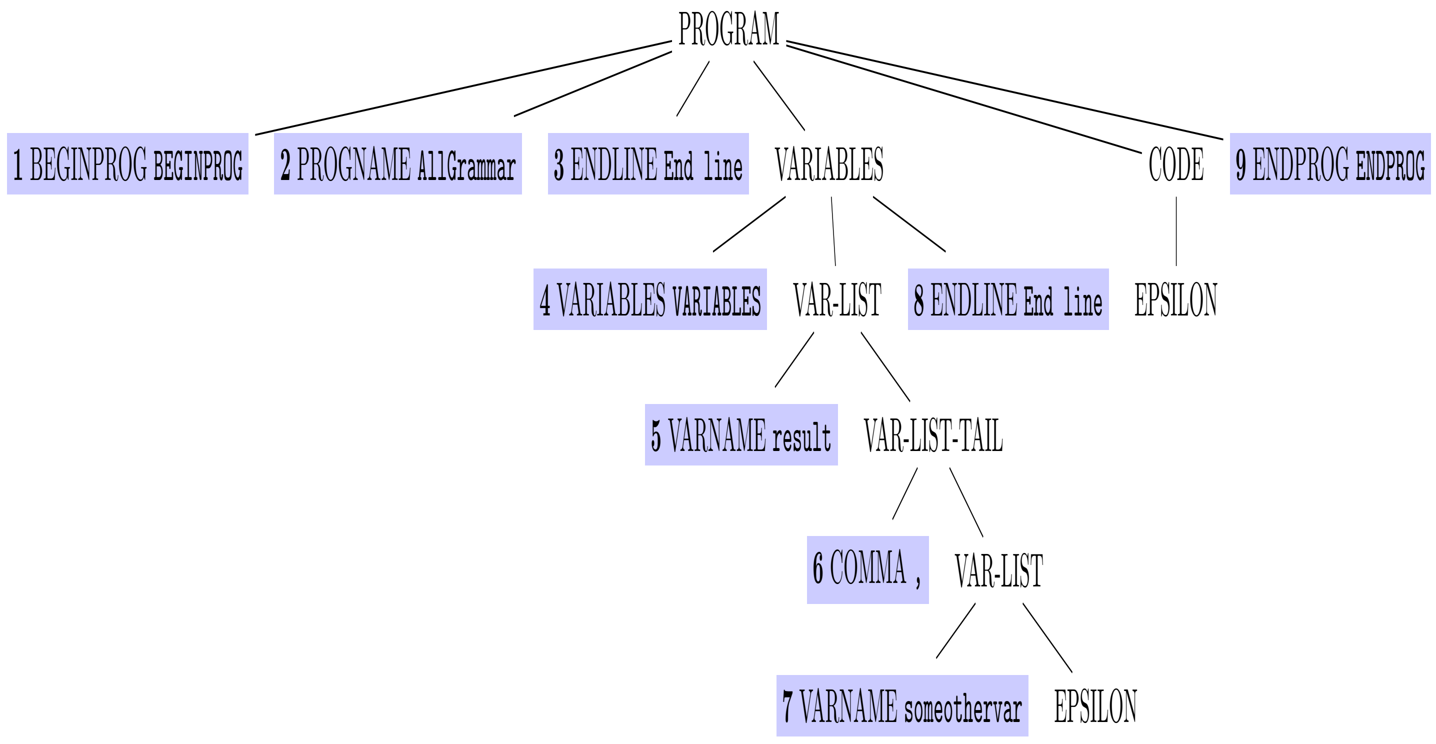

Figure 1: Parse tree example

7 Command Line arguments

7.1 -v - -verbose

In order to have a more verbose output of the left most derivation of the input string, the arguments **-v** or **- -verbose** can be passed to the command line, which will output all the rules along with their corresponding description about what rule has been applied:

```
1        by BEGINPROG [ProgName] [EndLine] <Variables> <Code> ENDPROG [EOS]
2        by <Variables> -> VARIABLES <VarList> [EndLine]
4        by <VarList> -> [VarName] <VarListTail>
...
```

7.2 -wt filename.tex

Additionally, to create a .tex file containing the parse tree described above, the argument "**wt filename.tex**" can be passed to the command line. If no filename is specified the program will create a file called parse_tree.txt for you. A nice message will be displayed into the console once the job is done.

```
Parse tree was created. Please check parse_tree.tex file.
```

7.3 -t

In order to show all the tokens given by the Lexical Analyzer, the argument "**-t**" can be passed to the command line.

```
    token: BEGINPROG      lexical unit: BEGINPROG
    token: AllGrammar     lexical unit: PROGNAME
```

7.4 -i

In order to show all the identifiers given by the Lexical Analyzer, the argument "**-i**" can be passed to the command line.

```
    Identifiers
    -----------

    i        26
    j        25
```

7.5 -h - -help

A help menu will be displayed by passing the argument "**-h**" or "**- - help**" in order to show the user the arguments that can be used.

```
    -v or --verbose Show the left most derivation tree detailed list.
    -t              Show the tokens list
    -i              Show the identifiers list.
    -wt [FILENAME]  Create a .tex file to see the
                    Parse Tree. Default file: parse_tree.tex.
```