# ASSIGNMENT #3: SYNTAX ANALYSIS

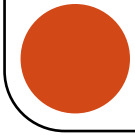
*BY*

*Craig Campbell (#50111)*
*And*
*Paul M. Baillot (#2273596)*

*Presented to Gregor Bochmann*

*For the course SEG-2106*

*University of Ottawa*
*2008-03-24*



## Contents

## Objective

The objective of this assignment is to create parser for a subset of the Pascal language. In order to complete the assignment, it will be necessary to create an LL(1) compliant grammar; it may be necessary to change it a bit to reflect the parts that we do not need to take care of. Afterwards, it will be required to create a set of FIRSTS and FOLLOWS and then to populate a parsing table. The next step after all this is done will be to implement a programmed version of the parser. The primary task of the automated parser will be to validate the language. The secondary task of the parser and its minimum requirement for this will be to evaluate "constant" integer expressions and "constant" boolean expressions.

## Part One: Defining The Grammar On Paper

### Grammar for the Subset of Pascal

**Legend**
BLUE : literal terminals
**DARK RED**: terminals

Items in dark red will be defined explicitly at the bottom of the list to explain exactly what they contain and represent. For clarity, some lines were placed one under the other. In this language, all spaces, tabs, carriage returns, line feeds and comment lines will be considered as blank spaces. *(The explicit definition of a blank space is given at the bottom of this list, but serves only as a tool to help filter different types of blank spaces and is not part of the subset of this language)*

**PROGRAM** → program **ID**;
**COMPOUND_STATEMENT**
.

**COMPOUND_STATEMENT** → begin
**STATEMENT_LIST**
end

**STATEMENT_LIST** → **STATEMENT**; **STATEMENT_LIST** | ∑

**STATEMENT** → **VARIABLE ASSIGNOP EXPRESSION**
| **COMPOUND_STATEMENT**
| while **EXPRESSION** do **STATEMENT**

**EXPRESSION** → **SIMPLE_EXPRESSION EXPRESSION'**

**EXPRESSION'** → **RELOP SIMPLE_EXPRESSION** | ∑

**SIMPLE_EXPRESSION** → **TERM SIMPLE_EXPRESSION'**

**SIMPLE_EXPRESSION'** → **ADDOP TERM SIMPLE_EXPRESSION'** | ∑

**TERM** → **FACTOR TERM'**

**TERM'** → **MULOP FACTOR TERM'** | **∑**

**FACTOR** → **VARIABLE** | **NUM** | ( **EXPRESSION** )

**VARIABLE** → **ID**

Explicit definitions of the mentioned literals:

**ID** → [A-Za-z]+[A-Za-z0-9]*

**NUM** → [0-9]+

**RELOP** → "=" | ">" | "<" | ">=" | "<=" | "<>"

**ADDOP** → + | -

**MULOP** → * | /

**ASSIGNOP** → :=

Special case (not part of language). The following line explicitly defines what blank spaces and comments are.

**BLANKSPACE_COMMENTS** → <space> | Carriage return | Line Feed | Tab| "{"([^1}]1)*"}"

## *FIRST And FOLLOW List*

Note: As stated earlier, the following are considered literals for simplification. They are explicitly defined in the "Grammar for the Subset of Pascal" section; they include: ID,NUM,RELOP,ADDOP,MULOP,ASSIGNOP.

FIRST (PROGRAM) → {program}
FIRST (COMPOUND_STATEMENT) → {begin}
FIRST (STATEMENT_LIST) → {FIRST(STATEMENT),∑} = {ID,NUM,(,begin,while,∑}
FIRST (STATEMENT) → {FIRST(VARIABLE),FIRST(COMPOUND_STATEMENT),while} = ...
= {ID,begin,while}
FIRST (EXPRESSION) → {FIRST(SIMPLE_EXPRESSION)} = ... = {MULOP,∑}
FIRST (EXPRESSION') → {RELOP,∑}
FIRST (SIMPLE_EXPRESSION) → {FIRST(TERM)} = ... = {MULOP,∑}
FIRST (SIMPLE_EXPRESSION') → {ADDOP,∑}
FIRST (TERM) → {FIRST(FACTOR)} = ... = {ID,NUM,(}
FIRST (TERM') → {MULOP,∑}
FIRST(FACTOR) → {FIRST(VARIABLE),NUM,(} = ... = {ID,NUM,(}
FIRST(VARIABLE) → {ID}

FOLLOW (PROGRAM) → {$}
FOLLOW (COMPOUND_STATEMENT) → {.,FOLLOW(STATEMENT)} = ... {.$, ;}
FOLLOW (STATEMENT_LIST) → {end}
FOLLOW (STATEMENT) → {;}
FOLLOW (EXPRESSION) → {FOLLOW(STATEMENT),do,)} = ... = {;,do,)}
FOLLOW (EXPRESSION') → {FOLLOW(EXPRESSION)} = ... = {;,do,)}

FOLLOW (SIMPLE_EXPRESSION) → {FIRST(EXPRESSION'),FOLLOW(EXPRESSION')} = ...
= {RELOP,FOLLOW(EXPRESSION'), ;,do,)} = {RELOP,;,do,)}
FOLLOW (SIMPLE_EXPRESSION') → {FOLLOW(SIMPLE_EXPRESSION)} = ... = {RELOP,;,do,)}
FOLLOW (TERM) → {FIRST(SIMPLE_EXPRESSION')} = ... = {ADDOP,FOLLOW(SIMPLE_EXPRESSION')}
= {ADDOP,RELOP,;,do,)}
FOLLOW (TERM') → {FOLLOW(TERM)} = ... = {ADDOP,RELOP,;,do,)}
FOLLOW (FACTOR) → {FIRST(TERM')} = ... {MULOP,FOLLOW(TERM')} = {MULOP,ADDOP,RELOP,;,do,)}
FOLLOW (VARIABLE) → {ASSIGNOP,FOLLOW(FACTOR)} = ... {ASSIGNOP,MULOP,ADDOP,RELOP,;,do,)}

## *Parsing Table*

The parsing table is too wide to display inside one page, this is why it will be split in two halves for higher clarity. Although they have been pasted as bitmaps to fit perfectly inside the borders of this page, the excel file will be provided as a separate file for your convenience (and is not split in half).

| | program | ID | ; | begin | end | . | while | do |
|---|---|---|---|---|---|---|---|---|
| PROGRAM | PROGRAM > program ID; COMPOUND_STATEMENT .$ | | | | | | | |
| COMPOUND_STATEMENT | | | | COMPOUND_STATEMENT > begin STATEMENT_LIST end | | | | |
| STATEMENT_LIST | | STATEMENT_LIST > STATEMENT; STATEMENT_LIST | | STATEMENT_LIST > STATEMENT; STATEMENT_LIST | STATEMENT_LIST > ∑ | | STATEMENT_LIST > STATEMENT; STATEMENT_LIST | |
| STATEMENT | | STATEMENT > VARIABLE ASSIGNOP EXPRESSION | | STATEMENT > COMPOUND_STATEMENT | | | STATEMENT > while EXPRESSION do STATEMENT | |
| EXPRESSION | | | EXPRESSION > ∑ | | | | | EXPRESSION > ∑ |
| EXPRESSION' | | | EXPRESSION' > ∑ | | | | | EXPRESSION' > ∑ |
| SIMPLE_EXPRESSION | | | SIMPLE_EXPRESSION > ∑ | | | | | SIMPLE_EXPRESSION > ∑ |
| SIMPLE_EXPRESSION' | | | SIMPLE_EXPRESSION ' > ∑ | | | | | SIMPLE_EXPRESSION ' > ∑ |
| TERM | | TERM > FACTOR TERM' | | | | | | |
| TERM' | | | TERM' > ∑ | | | | | TERM' > ∑ |
| FACTOR | | FACTOR > VARIABLE | | | | | | |
| VARIABLE | | VARIABLE > ID | | | | | | |

| | assignop | relop | addop | mulop | num | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|
| PROGRAM | | | | | | | | |
| COMPOUND_STATEMENT | | | | | | | | |
| STATEMENT_LIST | | | | | STATEMENT_LIST > STATEMENT; STATEMENT_LIST | STATEMENT_LIST > STATEMENT; STATEMENT_LIST | | |
| STATEMENT | | | | | | | | |
| EXPRESSION | | | | EXPRESSION > SIMPLE_EXPRESSION EXPRESSION' | | | EXPRESSION > ∑ | |
| EXPRESSION' | | EXPRESSION' > RELOP SIMPLE_EXPRESSION | | | | | EXPRESSION' > ∑ | |
| SIMPLE_EXPRESSION | | SIMPLE_EXPRESSION > ∑ | | SIMPLE_EXPRESSION > TERM SIMPLE_EXPRESSION' | | | SIMPLE_EXPRESSION > ∑ | |
| SIMPLE_EXPRESSION' | | SIMPLE_EXPRESSION' > ∑ | SIMPLE_EXPRESSION' > ADDOP TERM SIMPLE_EXPRESSION' | | | | SIMPLE_EXPRESSION' > ∑ | |
| TERM | | | | | TERM > FACTOR TERM' | TERM > FACTOR TERM' | | |
| TERM' | | TERM' > ∑ | TERM' > ∑ | TERM' > MULOP FACTOR TERM' | | | TERM' > ∑ | |
| FACTOR | | | | | FACTOR > NUM | FACTOR > ( EXPRESSION ) | | |
| VARIABLE | | | | | | | | |

## Part Two: Implementation Of The Parser

### Assumptions, Constraints And Features!

**Assumption #1:**

The first part of the assignment states:

*program* →
**program id** ( *identifier_list* ) ;
*declarations*

However the homework later informs us that we can drop support for "identifier_list" (In number 1 of the "Your Task" section).

Because of this, we have dropped both the "identifier_list" as well as the surrounding parenthese in the grammar highlighted on top. However, to maintain backwards compatibility in case your test code contains parentheses', the implementation will support both the "program ID;" and the "program ID();" formats (with or without the parentheses).

**Assumption #2:**

The following has been defined in the grammar for our language:

**VARIABLE ASSIGNOP EXPRESSION**

An expression can either be a "boolean" or an "integer" in this case. However to simplify the implementation, the assignment suggested to only support the "integer" type for assignments.

In order to respect our chosen grammar and to maintain compatibility with more complicated code, our implementation will allow a boolean to be assigned to an integer type. In this case, and if evaluation of the expression is possible, the boolean will be converted to its integer value of "0" for false and "1" for true (FEATURE!).

**Assumption #3:**
The following has been defined both in our version of the grammar and the suggested one:
while **EXPRESSION** do **STATEMENT**
The while "expression" though *should* be a boolean expression, but in its currently form will allow both boolean and integer expressions to be defined inside the "expressions". Once again in order to satisfy our chosen grammar and to maximize the capability of the parser, both integer and boolean types will be allowed in this "EXPRESSION". However, when an integer value is detected, a "1" will mean true and a "0" will mean "false". If it's possible to evaluate the integer expression, the boolean equivalent will automatically be applied instead (for example, if (1+1) is the expression, it will be replaced by "true" and if (1-1) is placed, it will be replaced by "false").

**Assumption #4 and Feature:**
It was unclear if the mathematical operations that must be supported by the implementation of the parser have to just be able to perform a single operation separated by a single binary operator such as:
myVariable := 1+2;
Or if long operations need to be supported such as:
myBetterVariable := 5-2*2/2+(1+1);
In order to comply with the chosen grammar, the complex operations will be supported. If it wasn't required, then it's a FEATURE ☺ !

**Assumption #5 and Feature:**
It wasn't 100% clear if we needed support for comments using the "{" and the "}" that pascal supports. It's mentioned in number "1" of the "lexical conventions" page but not anywhere else. For the purpose of being able to have comments inside the programs, that feature will be supported (comments will be considered blank spaces as stated earlier). If it wasn't required, then it's a FEATURE ☺ again!

**Constraint #1:**
It's not *really* a constraint, but more of a clarification. The program supports substractions and negative values. However, the grammar suggested:

*simple_expression* →
*term*
| *sign term*
| *simple_expression* **addop** *term*

Ok, we have a simple_expression ADDOP term. Then we have that a factor can be a NUM, and that a NUM can be a sequence of "DIGIT"'s, but the DIGIT's recommended definition is:

**letter** → [a-zA-Z]
**digit** → [0-9]
**id** → **letter** ( **letter** | **digit** )*

It is quite definitely [0-9] where a NUM can be [0-9]+ … This makes it a bit difficult to have lets say:
Bob:=-5; {doesn't work and undefined in the grammar}
This line does not really work with the suggested grammar and the grammar that has been chosen. However, there is a work around for this, just use:
Bob:=0-5; {works fine}
And that will work ☺ In Pascal, you used to have to do stuff like that sometimes for instance with "real" numbers, if you wanted to a ssign a "real" the value of 15, you had to write "15.0", so I guess in essence, we could call this a typical Pascal workaround.

**Feature!:**
A small prediction mechanism will be present in the implementation of the parser. It will do a few things.
First, it will attempt to detect division by zero's and abort the parsing.

Secondly, let's take the following code for example:
Var1:=1+1; {This clearly equals to 2.  This variable will be SAVED in the symbolTable}
Var2:=blah+blah2; {Well, it won't be evaluated at this time…}
Var2:=Var1+1; {This will actually be evaluated!  Because the parser is 100% sure that Var1 is absolutely 2 at this point.}
So Var2 will actually be evaluated as Var2:=3 because Var1 was successfully evaluated beforehand.  But I know what you're thinking, so let's take this example instead:
Var1:=1+1; {Var1:=2 now}
Var3:=blah;
Var1:=Var3; {Oh no!  We just crushed Var1 with an expression that cannot be evaluated at this point}
{it's okay though, Var1 will actually now be removed from the symbolTable because of this}
Var2:=Var1; {This expression will not be evaluated because the parser is no longer sure of the value of Var1}
Var1:=5; {Ah, we know what Var1 is again, this value will now be stored once more in the symbolTable}
Var2:=Var1; {This expression will actually be evaluated now and Var2:=5}
So what we are trying to say is that the parser will actually try its best to evaluate expressions containing variables when it actually knows with a 100% certainty what the value must at that point in time, which… could lead into eventual optimizations in the size and speed of the compiled code.

**Feature!:**
You no longer need a <space> at the end of the file or even an <empty line> for the parser to work correctly.  You can literally have the file end with "end." and that's it, and it will work just fine.

## Discussion

This homework was particularly hard because it took multiple concepts we learned either in class or in the lab and increased the complexity and difficulty by a logarithmic leap. In a previous lab, we had already seen how to take possibly ambiguous grammars and convert them either in an unambiguous LL(1) or LR(1) notation. However, the amount of lines in this grammar made it exceptionally difficult to spot instances of infinite recursion or ambiguousness in the grammar. Whereas in the classroom it was possible to sometimes use a "intuitive" approach, it wasn't possible this time. You needed to go line by line, finding left recursions, finding terminals that were multiple times at the start of a line for one non-terminal and finally finding ways of possibly simplifying the whole thing by either removing entire non-terminals like the infamous "optional_statement" that we removed or by grouping a terminal definition as a single terminal, such as ID and NUM. Although it doesn't look like it, an extensive amount of time was spent just looking at the suggested grammar, at the grammar we put forth, and absolutely ensuring that the grammar was perfect, before moving along to the FIRST and FOLLOW list. We wanted to make sure it was okay, because any errors in the grammar could result in a serious waste of time in the fabrication of the FIRST and FOLLOW list and as well as the parsing table and could lead to potentially disastrous set backs in the implementation. Once we went over the grammar about 100 times and were sure that it was okay, we moved on to the FIRST and FOLLOW list.

The FIRST and FOLLOW list was really just like doing it for the first time again. This was done in the lab, but this time was so much more complex due to the immensity of the grammar. Although tricks were used in the lab to speed up the process, absolutely no tricks were used this time, it was a line by line process. If you look at it right now, you will notice some lines that are written like this: "= … =". These lines mean that what precedes them couldn't be evaluated if you will right away, so they were skipped, we went to the next line and repeated the process line by line. Once we got to the bottom of either the FIRST or the FOLLOW list, we then climbed back up, now filling the right most part of the "= … =" with the now available information. Once this was done, it was reviewed again twice or three times, not as much as the grammar itself because the FIRST and FOLLOW list is so difficult to read in the first place, hence the color coding we added to add readability. The parsing table was an interesting feat just as the grammar and the FIRST/FOLLOW list were themselves. What we can say about it is that unlike the grammar and the FIRST/FOLLOW list, we thought we knew how to build it after having seen it in class, until we tried and really we had no idea. This meant studying once more using various sources exactly how such a parsing table is built. After a few hours or raw training, we were ready to build the parsing table and went through it with ease (and we wrote down the algorithm we went through in case there is a next time!).

It was now time for the big one, the implementation of an automated parser for our grammar, there is so many things that could go wrong with it, there are so many ways that serious problems would arise and set us back, but it was time. What we were most glad about, was to have a solid grammar, a grammar we could follow without coming the next day and saying, ~~oops, I did it again~~ there was a mistake. Now we did try a couple things quickly to know which direction we would take, we tried playing with the "import java.util.regex.*;" class and really, we didn't like it for the purpose of this parser. We thought about having an intermediate file using linux's lex as the basis for creating the file containing the lexeme, but the lex code just crashes under both Windows XP and Vista and was just going to cause more headache that good. We decided that to implement the parser, we would use lab 6's code as a basis for the parser. First thing's first, we needed to get the Lexer class up to par, it was missing an arm and a leg when we started. We worked on the Lexer class, adding all the missing terminals, making all the necessary changes, improvements (such as the comments feature!) necessary to prepare it to work with the upcoming Syner class. Some minor changes included making all the constants STATIC instead of not static, why so? Because in their non-static form it was impossible to use them in "case" statements because the java compiler complained that they weren't true constants. The getNextToken() procedure was obviously the biggest change, from supporting all the new terminals to handling the EOF exception differently to avoid needing blank spaces at the end of the source file (catch (Exception e) {cCharReadFromFile = '\0'; //Softly allows an EOF to occur without throwing an error).

We then made our way to the Syner class, it was just what we didn't need for our grammar, there wasn't much we could re-use easily, without having to refactor, mangle and otherwise cause serious deformations of the source the code. So what we did instead is we deleted absolutely every parse<anything> functions without taking a second look back. We seriously modified the startAnalysis() procedure and also modified the way the errorMessage(String) procedure worked. Really the only thing we kept from that class was well, the class name, the constructor and three private variables. The reason why we did something as harebrained as that was because we believed we had a way of making this thing worked the way we wanted it to. We were going to make our own parse<Something> functions all over again, but strictly following our grammar so that if our grammar has a "PROGRAM" non-terminal, we would have a "parseProgram()" procedure, and that if we had a " EXPRESSION' " non-terminal, we would also have a "parseExpressionPrime()" procedure and that would all fatefully perform exactly (or as close as possible to) as their non-terminal counterpart in grammar land. Once we decided that this is how it would be then this is what we did, each procedure built upon another, if one parse procedure fails, so must the other. Parse by parse they were written and tested individually. Up to the parseExpression, we did not need to think about how we would solve the "expression evaluation" problem, expressions much more complicated than in lab 6 (at least, if we wanted to accept what the grammar could accept). After testing the implementation so far and resolving minor issues, it was now time to determine how to do this. We decided to create four new private variables for the Syner class, a boolean that tells everyone if the expression has the possibility of being evaluated, a boolean that tells the procedures whether an integer number is being tracked at the moment or a boolean value, and then one integer and one more boolean, each holding their values so far in the evaluation and as needed. In order to have the capability of having complex expressions with many terms one after another and parentheses flying left and to the right, we added temporary variables in some of the procedures (the ones doing the calculations), the values of the classes 4 private variables would be saved temporarily, and the procedure would then verify and perform the next evaluation in line, one by one, and then add,substract,divide andmultiply them in order afterwards as the procedures would roll back one by one. Problems we had with the evaluation was stuff like the procedure placing numbers on the opposite signs of the signs, for example, instead of doing 5/2, it would do 2/5, it was fine for additions but didn't' work so well for substractions and divisions. Bigger problems included the special cases we talked about in the assumptions earlier about what to do in certain undocumented cases. We really wanted to push this program further against it's limits so if we saw a way we could make a procedure do more than we really needed, we just did, because in the end it was going to take just as long either way, we were already knee deep in the code and it was do or die.

In conclusion we believe that all the objectives for this assignment were met. The way we chose to attack the implementation worked as intended. We clearly saw that theory can look easy and straightforward in class, and that in reality, when you have 10 times the amount of data to work with that what seemed easy can become times and times more complex.