Pascal Compiler Final Report

By: Christopher Cartagena

Introduction:

In these two projects, the goal was to first perform type and scope checking. This would be done by adding semantic decoration to the existing syntax structure, that was created in the previous project. In the second project the memory of all the variables was computed, this was done being wary of the scope, so the memory numbering was restarted after every new scope and resumed at the close of the scope.

Methodology:

In order to get started talking about project 3, the previous projects should be explained in some small detail. The methodology for the first project consisted of creating several finite state machines by paper first to see ways in which to parse for tokens. After defining the state machines, several macros define statements were added to the header file. These defined numbers that specified a type, or the attribute for a token. The state machines where then created, and then tested using several text files to see that they were parsed correctly, and that the program was able to catch all lexical errors that were implemented. The way the next project was implemented was to start out by performing the following massaging operations, these consisted of the following:

1. Removal of ambiguity (except the dangling else ambiguity)

2a. Elimination of nullable productions

2b. Elimination of immediate and deep left recursion

3. Left factoring

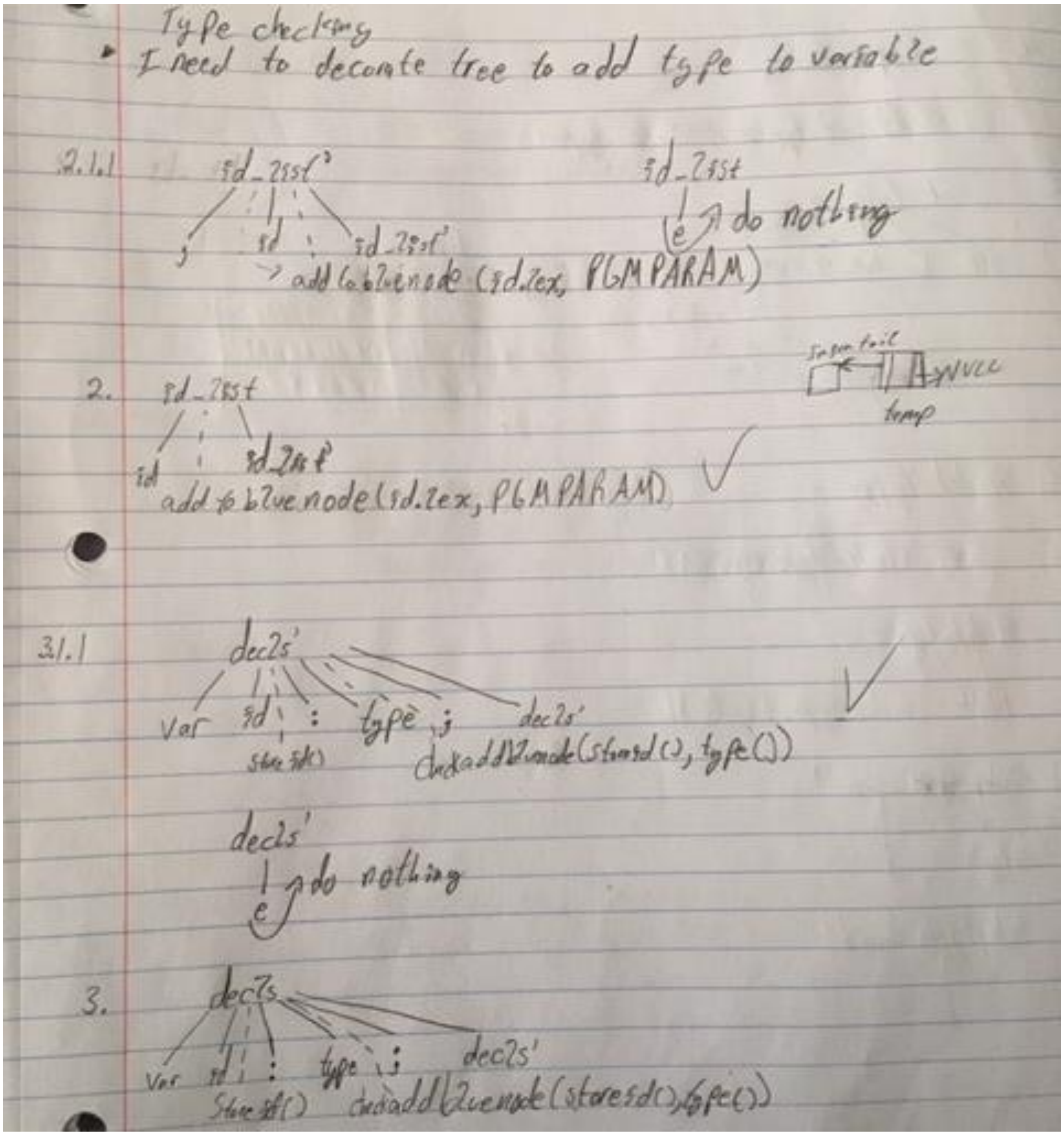
The first operation was performed in a word document, and all the eligible grammars were massaged. The second was done by copying from the resulting grammar of the first, and then copied into a new word document, and then all the eligible grammars were massaged. The third operation was done by copying from the resulting grammar of the second, and then copied into a new word document, and then all the eligible grammars were massaged. The fourth operation was done by copying from the resulting grammar of the third, and then copied into a new word document, and then all the eligible grammars were massaged.

After the final massaging operation was completed the first and follow of all the variables was found and stored into an excel file. Following this procedure, the parse table of the grammar was created, and stored into an excel file. A python script was then created that would read in the excel file and output the recursive decent parser in C syntax. The resulting syntax analyzer was then incorporated into the previously created Lexical Analyzer.

The process that was followed in project 3 was to decorate by hand, this was done by creating parse trees of all the grammar, and then decorating using paper and pencil. I first started with declaration processing, as it seemed it would be the easiest to implement, I then continued to type checking. In order to do this, I had to start from the bottom up, because of the way the grammars call itself, this means the last symbols in the grammar to be called would be at the very bottom. Scope checking was the final decoration to be added to the grammar.

In order to add

The following are images of the results of the hand decorations done.



4. type

stndrd_type type.t := stndrd_type()

type

arr [num_1 .. num_2] of stndrd_type

check num(num_1, num_2)
both ints
and num_1 < num_2

type.t :=

S.type	type.t
INT	INT ARRAY
REAL	REAL ARRAY
ERR	ERR*

5. stndrd_type

INT stndrd_type.t := INT

stndrd_type

REAL stndrd_type.t := REAL

8.1.1 subprgrm_head'

args ;

subprgrm_head'

;

8. Subprgrm_head

Procdr id ; Subprgm_head

Check add greennode (id.Lex, PGNAME)

9. args

(Param_lst)

10.1 Param_lst'

; id ; : type ; Param_lst'

storeid() checkaddbluenode(storeid(), type())

↳ if success add to function args

Param_lst'

e → do nothing

10. Param_lst

id ; : type ; Param_lst'

storeid() check add bluenode(storeid(), type())

↳ if success add to function args

1.1.1.3.1 Prgrm''

ε sbprgrm_decls Cmpnd_stmnt .

1. Prgm

Prgm id (id_list); prgm'

check_add_green_node(.id.lex, PGPARAM)

11.1.1.1. Cmpnd_stmnt'

opt_stmnt end close_scope

Cmpd_stmnt'

end close_scope

11. Cmpn_stmnt

new_scope begin Cmpnd_stmnt

22.4.1 factor'

[express]

$f.i = id.type$

$f.i$	express	$f'.t$
AINT	INT [illegible]	INT ERR
AREAL	INT REAL	REAL ERR
AINT AREAL	ERR*	ERR
ERR*	ERR*..	ERR
	Bool	ERR

factor'

ε $f'.t = f'.i$

22. 1 factor

id factor'

$f'.i = id.type$ $f.t = factor(id.type)$

2 factor

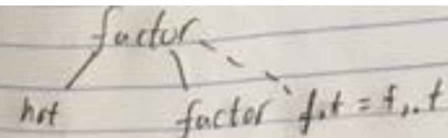
num $f.t = num.t$

3 factor

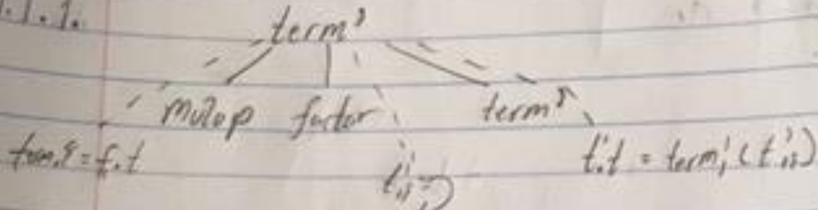
(express)

$f.t = express()$

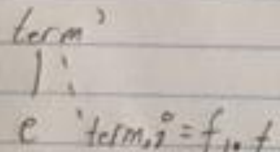
9



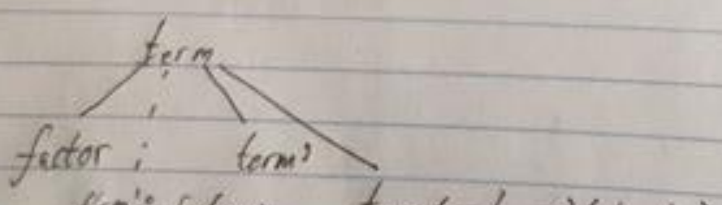
21.1.1.



t'_i	t_i	f.t	mulop
INT	INT	INT	*, /, div, mod
REAL	REAL	REAL	
ERR	mismatch		
ERR	ERR	ERR	
BOOL	BOOL	BOOL	AND
ERR	mismatch		
ERR	ERR	ERR	



21.



20.2.1 Simp-express'

addop term Simp-express' $SE'.t = SE'_2$

$Se.i$ get addop and term

$Se_2.i$ =

$Se.t$	$Se.i$	term	addop
BooL	BooL	BooL	or
INT	INT	INT	+, -
REAL	REAL	REAL	+, -
ERR*	ERR*	ERR	ERR
ERR	Anything Else		

Simp_Express'

e

$Se'.t = Se'.i$

20 Simp_express

Sign term Sim_express' $SE.t = SE'.t$

$SE.i$ = term

Simp_express

term Simp_express' $SE.t = SE'.t$

$SE.i$ = term

19.1.1 express'

relop simp_express express'

express'.i get relop()

express'.i	simp_express	express'.s
INT	INT	Bool
REAL	REAL	BOOL
ERR	OR ERR	ERR*
mixed	mode not allowed!	ERR

express'

e express'.s = express'.i

19. express

simp_express express'

express.s = express'(simp_express)

18.1.1 express_list'

; express express_list'

add to list of function parameters

express_list'

e

18. express_list

express express_list' — type check function and parameter

add to list of function parameter

16.1.1

Variable'

[express]

variable'.i=id

variable'.t=

id	express	type
A INT	INT	INT
A REAL	INT	REAL
ERR	ERR	ERR*
(ANY type)	Error	ERR

Get relative variable

Variable'

ε variable.t=variable'.i

16 Variable

id Variable'

variable'.i=id

Variable.t=variable'(variable'.i)

17.1.1 procdr_stmnt'

(expresslist')

Check function exist()

procdr_stmnt'

ε

17. procdr_stmnt

call id procdr_stmnt'

procdr_stmnt'.i=id

check procedure name exists

15. stmnt'

else stmnt

stmnt

e

14. stmnt

variable : assignop express

get variable type

get expression type

Action	variable	express
ERR*	ERR or	ERR
same		same
ERR		else

stmnt

procdr_stmnt

nothing

stmnt

cmpnd-stmnt

nothing

stmnt

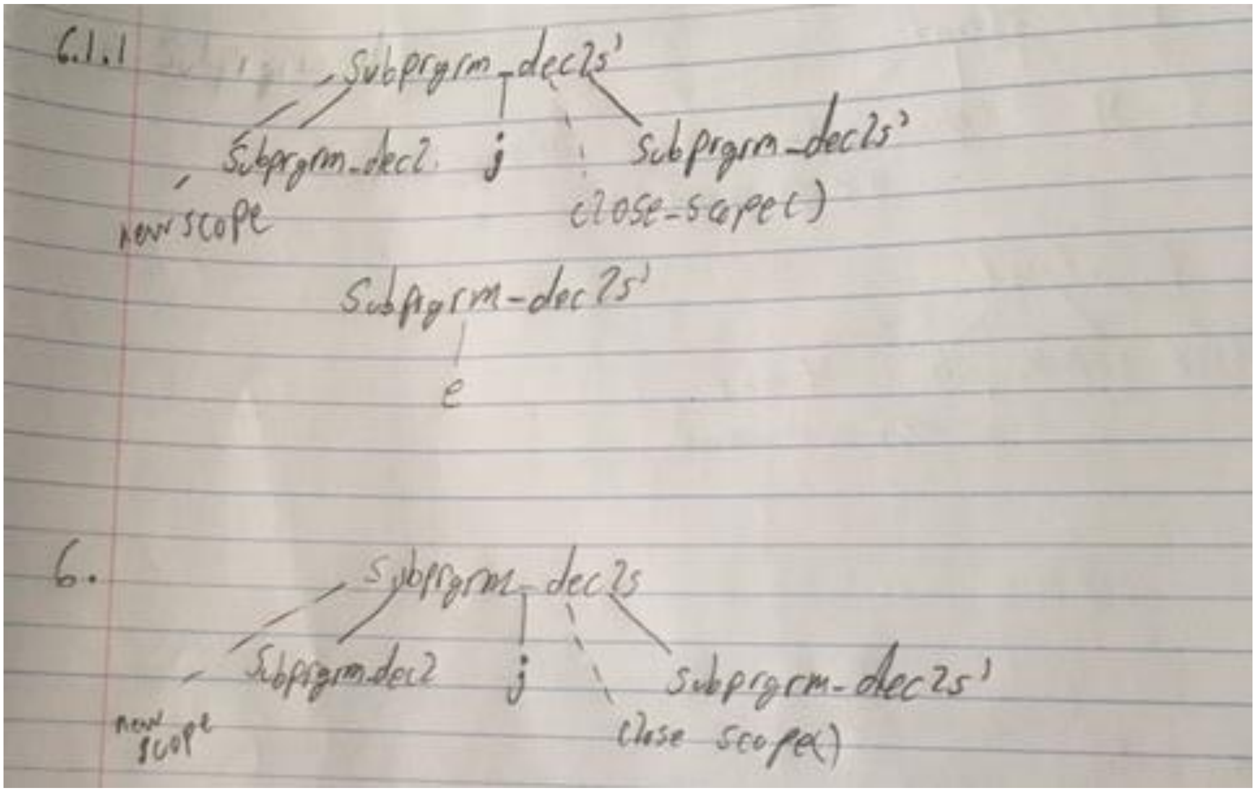
if express then stmnt stmnt'

is expression bool

stmnt

while express do stmnt

is expression boolean



Implementation:

One of the first things that needed to be created for the project were the structs that were to be used to define a token. The token was defined using a struct and a union. The struct defined the line number that the token was to be found at, the lexeme of the token, its type and the union attrib, which defined the attribute, finally there was a pointer to the next token. The attrib union defined the attribute of the token as either an integer value which would be the case for most tokens, or a pointer to an integer which would be the case for an id, where the pointer was to the memory address where the id was stored. The lines were counted using a global variable that incremented every time a new line was read in. To keep track of the reserved words for the pascal language, a text file with all of them was created, and then fed into a linked list. Then to check for a match the linked list was iterated over, and every reserved word compared to the pulled lexeme. The reserved words that were used were the following:

if	**else**
then	**while**
program	**do**
var	**or**
array	**div**
of	**mod**
integer	**and**

real	**not**
procedure	**call**
begin	
end	

The first machine that was implemented was the whitespace machine, which basically just moved the index of the buffer that contained the current line, past the white space, tabs, or new line characters. The next machine that was implemented was the id state machine which would read in an alphabetical character first, and then read in the following alphanumerical characters that followed. This state machine capped off at 10 characters and generated an error if it saw a longer id.

The next machine was the relational operator machine, this machine first tried to look for the longer relational operators like the not equals, greater than or equals, or less than or equal to first. Then after that it looked for the rest of the relational operators. The machine that looked for numbers, first looked to see if the number was a long real, if it did not find it, then it checked if it was a short real, and if it did not find it then assumed that it was an integer. This machine then passed the parsed number to separate smaller machines that handled each case where it was either an integer, a short real, or a long real.

Aside from the above state machines, other function where also implemented in order to check for lexical errors, or add a token to the token linked list, to covert the macros used to string for details, or to print the token, and listing output.

After the massaging operations were performed in pencil and paper, a python script was created to write the recursive decent parser. The way this was done was by first importing the excel file the parse table was written in. The first row was stored to signify all the available terminals. The first column was also stored, and these were all the variables. Furthermore, a file which contained all the terminals and their associated macro defined numbers was also used. This was so that case statements could be used to reference the defined numbers and as a means of cross referencing them in the python script.

The python script then goes through all the variables parse table values, and checks if any of they are terminals. If they are not terminals then the script will call the function, if not then a match statement is called with the parameter of the terminal. All the functions that are used for the variables have been defined at the very top of the file. The synch set is then used in the default in order to synch any syntax errors that could be encountered. Any syntax errors that are generated in the middle of processing are stored into a linked list for syntax errors, and their line numbers are also stored.

Once the parsing has been completed, either as a result of the end of file token being correctly encountered, or a result of too many syntax errors, the listing file is created. The way in which the listing file is created is that for each line that is printed, the linked list for the syntax analyzer is traversed to see if there were any syntax errors at that line, if so, it will print them out.

In order to implement the semantic decorations in the current syntax analyzer, they were added in between function calls. Furthermore, several of the functions that were called in the program were changed from void to returning an integer. This was to incorporate the attribute behavior that many of them had. Not just that, but for several of the functions, a parameter had to be added to its function definition, this was in order to mimic the inherited attribute behavior that they would have.

Furthermore, a new file was created for this project, that would have all the functions necessary to perform the semantic analysis. In order to perform the table checking behavior of the grammar, several if statements were used. The errors that would be reported in the grammar were only reported one time, to make sure there would not be a cascading of errors that were all trying the report the same one.

In order to perform the scope checking part of the behavior what needed to be done was that there would have to be a close scope function that would close the parameters currently in the infrastructure that would be used to check for scope. To assign for memory adding function was added almost alongside of the scope checking. This memory was done by having a global memory counter, and any time a new variable was declared then it would be continued. Also, anytime that a new scope was created then it would be reset to 0. This memory allocation had the added the feature that it would restart the numbering from the last place that the last scope had left it off. This is because a stack was used to store the previous memory numbering before it was reset to 0, when a new scope was reached.

Discussions and Conclusions:

The first part was built in a way that a further machine could receive all the tokens in a linked list, which makes it helpful for implementing the second part of this project, in which a syntax analyzer is used. The conclusion of this project was a program that could extract tokens related to the pascal language from a text file, and furthermore be able to identify some of the lexical errors that could occur and give details about them.

The second part was built using a recursive decent parser from a massaged LL1 grammar. The massaging operation was probably the harder part of this assignment, while not very complex in the process, it was a bit tedious. Once this was done however, the recursive decent parser code was easily created with the help of a python script that was created. There was also a debugging portion that was done, to find a couple of problems mostly with the way that the parse table in the excel file, this was mainly due to incorrect grammar being placed, basic human error. The resulting syntax analyzer is now ready for semantic decoration.

The final part of this project was to declaration processing, type/scope checking, and memory allocation. This was all done by decorating the previous grammar with semantic decorations. Furthermore, some of the previous functions had to be changed to return integers, and this was due to having to return attributes, and to pass inherited attributes.

Appendix I: Sample Inputs and Outputs

Sample program:

```
prgrm example(one,two);
var first : int;
var second : real;
var third : int;
procdr FirstTest (primary: int; secondary: real);
var fourth : int;
var fifth : real;
begin
  if fourth = 0 then
  fourth := primary
end;
procdr Second(tertiary : int; quarteary: arr [0 .. 5] of real);
var seventh : real;
begin
  seventh := second + second
end;
begin
  first := third
end .
$
```

Listing Output:

```
1     prgrm example(one,two);
2     var first : int;
3     var second : real;
4     var third : int;
5     procdr FirstTest (primary: int; secondary: real);
6     var fourth : int;
7     var fifth : real;
8     begin
9       if fourth = 0 then
10        fourth := primary
11     end;
12     procdr Second(tertiary : int; quarteary: arr [0 .. 5] of real);
13     var seventh : real;
14     begin
15       seventh := second + second
16     end;
17     begin
18       first := third
19     end .
20     $    Successfully parsed !
```

Appendix II:

Token Output:

Line	Lexeme	Token Type		Attribute	
1	prgrm	3	PROGRAM	0	NULL
1	example	20	ID	B1A0E0	NULL
1	(	80	OPEN_PARENTHESES	0	NULL
1	one	20	ID	B19ED0	NULL
1	,	82	COMMA	0	NULL
1	two	20	ID	B1A170	NULL
1	)	81	CLOSED_PARENTHESES	0	NULL
1	;	79	SEMICOLON	0	NULL
2	var	4	VAR	0	NULL
2	first	20	ID	B1A0F0	NULL

2	:	75	COLON	0	NULL
2	int	7	INT	0	NULL
2	;	79	SEMICOLON	0	NULL
3	var	4	VAR	0	NULL
3	second	20	ID	B1A120	NULL
3	:	75	COLON	0	NULL
3	real	8	REAL	0	NULL
3	;	79	SEMICOLON	0	NULL
4	var	4	VAR	0	NULL
4	third	20	ID	B1A1B0	NULL
4	:	75	COLON	0	NULL
4	int	7	INT	0	NULL
4	;	79	SEMICOLON	0	NULL
5	procdr	9	PROCEDURE	0	NULL
5	FirstTest	20	ID	B1AB80	NULL
5	(	80	OPEN_PARENTHESES	0	NULL
5	primary	20	ID	B1ADC0	NULL
5	:	75	COLON	0	NULL
5	int	7	INT	0	NULL
5	;	79	SEMICOLON	0	NULL
5	secondary	20	ID	B1ACD0	NULL
5	:	75	COLON	0	NULL
5	real	8	REAL	0	NULL
5	)	81	CLOSED_PARENTHESES	0	NULL
5	;	79	SEMICOLON	0	NULL
6	var	4	VAR	0	NULL
6	fourth	20	ID	B1AC00	NULL
6	:	75	COLON	0	NULL

6	int	7	INT	0	NULL
6	;	79	SEMICOLON	0	NULL
7	var	4	VAR	0	NULL
7	fifth	20	ID	B1AC40	NULL
7	:	75	COLON	0	NULL
7	real	8	REAL	0	NULL
7	;	79	SEMICOLON	0	NULL
8	begin	10	BEGIN	0	NULL
9	if	1	IF	0	NULL
9	fourth	20	ID	B1BF90	NULL
9	=	160	RELOP	124	EQU
9	0	7	INT	22	VALUE
9	then	2	THEN	0	NULL
10	fourth	20	ID	B1C020	NULL
10	:=	163	ASSIGNOP	0	NULL
10	primary	20	ID	B1C160	NULL
11	end	11	END	0	NULL
11	;	79	SEMICOLON	0	NULL
12	procdr	9	PROCEDURE	0	NULL
12	Second	20	ID	B1BF50	NULL
12	(	80	OPEN_PARENTHESES	0	NULL
12	tertiary	20	ID	B1C230	NULL
12	:	75	COLON	0	NULL
12	int	7	INT	0	NULL
12	;	79	SEMICOLON	0	NULL
12	quarteary	20	ID	B1C210	NULL
12	:	75	COLON	0	NULL
12	arr	5	ARRAY	0	NULL

12	[	84	OPEN_BRACKET	0	NULL
12	0	7	INT	22	VALUE
12	..	83	DOUBLE_PERIOD	0	NULL
12	5	7	INT	22	VALUE
12	]	85	CLOSED_BRACKET	0	NULL
12	of	6	OF	0	NULL
12	real	8	REAL	0	NULL
12	)	81	CLOSED_PARENTHESES	0	NULL
12	;	79	SEMICOLON	0	NULL
13	var	4	VAR	0	NULL
13	seventh	20	ID	B1D370	NULL
13	:	75	COLON	0	NULL
13	real	8	REAL	0	NULL
13	;	79	SEMICOLON	0	NULL
14	begin	10	BEGIN	0	NULL
15	seventh	20	ID	B1D570	NULL
15	:=	163	ASSIGNOP	0	NULL
15	second	20	ID	B1D450	NULL
15	+	161	ADDOP	71	ADD_SYMBOL
15	second	20	ID	B1D620	NULL
16	end	11	END	0	NULL
16	;	79	SEMICOLON	0	NULL
17	begin	10	BEGIN	0	NULL
18	first	20	ID	B1D680	NULL
18	:=	163	ASSIGNOP	0	NULL
18	third	20	ID	B1E110	NULL
19	end	11	END	0	NULL
19	.	78	PERIOD	0	NULL

20	$	200 EOF	0	NULL

Memory Allocation:

```
ID: first, TYPE: INT, MEMORY: 0
ID: second, TYPE: REAL, MEMORY: 4
ID: third, TYPE: INT, MEMORY: 12
ID: primary, TYPE: INT, MEMORY: 0
ID: secondary, TYPE: REAL, MEMORY: 4
ID: fourth, TYPE: INT, MEMORY: 12
ID: fifth, TYPE: REAL, MEMORY: 16
ID: tertiary, TYPE: INT, MEMORY: 0
ID: quarteary, TYPE: AREAL, MEMORY: 4
ID: seventh, TYPE: REAL, MEMORY: 44
```