**CS4013**

**Project 3 & 4**

A Semantics Analyzer and addressing scheme for a subset of the Pascal programming language

By Carson Ellsworth

0976

**Part 1)** **Introduction:**

For the third project in building a front-end compiler, I was assigned to building a semantic analyzer. Semantic Analyzer’s dissect the meaning of a program to analyze the type information and catch errors within the program based on the programming language specifications, this includes base type checking of variables as well as the scoping rules of the language. For the Semantic Analyzer to work correctly in my one pass implementation of the compiler, I built it upon the underlying structure set up for Syntax analysis in project 2.

The type checking done for this compiler involves six types, integers, reals, array of integers, array of reals, Booleans, and type error. The first four types are able to have variables and function parameters defined with such types. The way to have access to the Boolean type is to use some valid combination of the four using an expression that utilizes a relational operator. And the way to acquire a type error is to simply use an invalid combination of the types.

For the combination of the types to be valid a project parameter was declared so that no mixed mode arithmetic (MMA) would be allowed for the compiler. MMA is the arithmetic that deals with the type analysis of two different types given a single operator. For example, to analyze the proper type of the expression a:= 5 + 5.5; a compiler will need to know how to add two different types together. In this case the addition of a real and integer has been determined to be a real because this helps prevent loss of information.

To properly type check the pascal subset language I also would need to build a scoping system to account for variable and function creations within certain levels of the program, the scoping system helps determine what variable and it’s type is being referred to in a program if there are many instances of that same variable name used. This also allows for the catching of semantic errors of variables being used outside the bounds of where they are allowed to be called from.

Finally, this project also covers generating addresses for variables within local scopes. If the rest of this project is implemented correctly then doing this part is a breeze compared to the rest of the projects. For the project each local scope would calculate an address for the variables within it to use given their type and size. As per the given parameters an integer takes up four bytes and a real takes up eight bytes with their respective arrays taking up a multiple of four or eight depending on their length.

**Part 2)** **Methodology:**

In determining how to make type information flow I encoded all possible types as different integers and made all of the structures in project 2 return integers rather than void. I then worked on paper out how a majority of the information will be transported through the code. To do this I went back to the productions that I derived to make the ll(1) grammar and expanded out each production and its tail to get several trees of the same production.

for instance, one of factors productions was <factor> -> **id** <factor’>. On my paper I would have expanded this out to <factor> -> **id (**<expr\_lst>**)** | **id** | **id [**<expr>**]** from these expansions I then would create separate type tables. For example the table below shows how the information must flow

Table for the production <factor> -> **Id** **[**<expr>**]**

|  |  |  |
| --- | --- | --- |
| Id.t | <expr>.t | Factor.t |
| Int Array | Int | Int |
| Real Array | Int | Real |
| Not array | Int | ERR\*(1) |
| Array | Not Int | ERR\*(2) |

The ERR\* signifies a semantic error, in the case of the compiler, it would inform the programmer in case 1 that a non-array identifier cannot be indexed, and in case 2 it would inform the programmer that an array cannot be indexed with a non-integer type. There are other cases than just these two for this particular example, but the idea is quite simple for a majority of the other tables. There are some cases unique to the language specification. Such as the information passing for the statement procdueres and for function calls.

<stmt> -> <var> **assignop** <expr> | <cmpd\_stmt> | **if** <expr> **then** <stmt> <stmt’>

While an argument could be made for the type of the first procedure, it is unreasonable to assume any type to the totality of the next two procedures that could encompass all the possibilities present. Unless we use the void type to assign a correct statement and the error type to assign mal-formed statements. This solves the problem because now there is a known type to check for when evaluating statement procedures.

Ex. for <stmt> -> <var> **assignop** <expr>

|  |  |  |
| --- | --- | --- |
| <var>.t | <expr>.t | <stmt>.t |
| Int | Int | void |
| real | real | void |
| Array Int | Array Int | void |
| Array Real | Array Real | void |

Function calls were another tricky business, this is because a function can have any number of parameters of any type. This obfuscates the path to being able to properly check and see if a function is called correctly with parameters of the correct type. A solution that I found to solve the issue was to create a linked node system of variables (this will be discussed more in part 3). This system allows me to differentiate between functions, function parameters and normal variables. Which will allow for construction of a relatively easy searching algorithm to return the necessary information to the type checking system.

**Part 3)** **Implementation:**

The biggest part of this project was by far implementing the node structure to correctly setup variables with their type information. The first big challenge in doing so was to correctly create a structure that would allow for all the necessary type information to be held. A single node holds the information of its name, type, node type (function, function parameter, or variable), referred to as color, and pointers to other nodes in front (next), behind (prev), above (parent), or below (child) them.



fun1 next fun2 next var2



child



next



fp1 var1

This example of the node structure shows three colors of nodes, green for functions, yellow for function parameters and blue for regular variables

this type of node structure could be created from a segment of the pascal language as follows.

…

function fun1:integer;

begin

end;

function fun2 (fp1:integer):real;

var var1 : array [1..25] of integer;

begin

end;

var var2 : array [1..2] of real;

…

In the diagram above the nodes that have a next pointer, such as that of fun2 to var2 would also have a pointer going from var2 to fun2 using the structures prev (previous) pointer, the same is also true for nodes with child and parent relationships.

To properly implement the pascal scoping structure, I also needed to create functions that would allow for the traversal of going up and down scope easy. Going up scope luckily was simple. To properly go up scope I needed to find where the production for a functions end was located. This production happened to be in the sub declaration call as it deals with function definition.

<sub\_dec> -> <sub\_head> <sub\_decT>

All my function for going up scope had to do was look for the nearest parent node and make that the top of the stack.

Knowing when to go down scope, while it might sound like an easy task, was actually very complicated. This is because there are many options when it comes to the first variable declared in the scope of a function, this could be a function parameter, variable, nothing, or an error. To overcome this problem, I created a lock system that the compiler uses. When a variable is created it looks to see if there is an available lock for that scope, if so it will be created as a child node to the front node of the stack, however if the lock has been consumed by a variable already then the variable in creation will just stick itself as the next node to the front of the stack. This lock system was able to solve all of my problems in one solution which was amazing with the exception of a function not having any declared variables within its scope. Though to counter this there is a manual lock override done at the same place the scoping up of the function takes place. And with that covers an incredibly simplified explanation of the scoping system.

Type checking from this point on was quite straight forward, I just kept creating my tables to see which productions needed to return what type given the scenarios. For calling functions however it was a bit trickier because I needed to figure out how to account for the parameter issue. Using the scoping system though I used the productions <expr\_lst> and <expr\_lstT> to compare if the amount of parameters present were equal to the amount declared as well as checked their types for continuity. If there was no errors in the parameter calls, then the production would return back to <factorT> -> **(**<expr\_lst>**)** with type void, and because I had access to the functions return type stored, if type void was present then I could safely return the functions type.

After all the type checking was implemented, addressing was incredibly easy to get going. Because project parameters state that in each new scope the addresses are recalculated, starting at zero. It was really easy to create a global variable to hold the current address and accumulate the correct values based on type information. The error handling for this part required me to store a previously calculated version of the address, just in case an error occurred during variable creation the address can still be reverted back to a correct value.

**Part 4)** **Discussion and Conclusions:**

The type checking and scoping for this project was by far the most challenging part of creating the compiler. This required by far the most mental energy from me, but was also the most rewarding when it came to problem solving, and is by far the best and biggest programming project I have worked on. It took me quite a while to learn how to translate this idea of information flow to actual code that worked with my compiler. However once I started grasping the fundamental concepts at play I quickly was able to translate the type information from abstract on paper to actual working code. The error handling is by far the most complicated part of any software and should truly be regarded as an art. The fact that my compiler is not allowed to just crash when a dumb programmer puts in error filled code and still produce meaningful output is a true testament to the resilience that this software is required to have.

**Part 5) References:**

Aho, A., Sethi R., Ullman J. (1986) Compilers Principles, Techniques, and Tools. Reading, MA: Addison-Wesly

**Appendix I: Sample Inputs and Output:**

No syntax errors

program test (input, output);

var a : integer;

var b : real;

var c : array [1..2] of integer;

var f : array [25..45] of real;

var g : array [1..2] of integer;

function fun1 : integer;

function fun5 : integer;

begin

end;

begin

end;

function fun2(x: integer; y: real) : real;

var e: real;

var ab : array [1..2500] of real;

var c: integer;

function fun3 : integer;

var g: integer;

begin

end;

function fun6 : integer;

begin

end;

function fun7 : integer;

var g: integer;

begin

end;

function fun4(hello: array [1..2] of integer) : integer;

begin

while(a=hello[2]) do

begin

e:=e

end

end;

begin

b := e + 4.44;

b:= (a mod x) / x;

while ((a >= 1) and ((b <= e) or (not (a = c)))) do

begin

b:= 2.5E2

end;

fun2 := 2.5

end;

begin

b:= fun2(2,f[5]);

if (a>2) then a:= 2 else a := a+2;

if (b > 4.2) then a := c[a]

end.

With just semantic errors

program test (input, output);

var a : integer;

var b : real;

var c : array [1..2] of integer;

var f : array [25..45] of real;

var g : array [1..2] of integer;

function fun1 : integer;

function fun5 : integer;

begin

end;

begin

end;

function fun2(x: integer; y: real) : real;

var e: real;

var ab : array [1..2500] of real;

var c: integer;

function fun3 : integer;

var l: integer;

begin

end;

function fun6 : integer;

begin

l:= 75

end;

function fun7 : integer;

var g: integer;

begin

end;

function fun4(hello: array [1..2] of integer) : integer;

begin

while(a=hello[2.5]) do

begin

e:=c

end

end;

begin

b := e + 4.44;

b:= (a mod x) / x;

while ((a >= 1) and ((b <= e) or (not (a = c)))) do

begin

c:= 2.5E2

end;

fun2 := 2.5

end;

begin

b:= fun2(2,f[5]);

if (a>2) then a:= 2 else a := a+2;

if (b > 4.2) then a := c[a]

end.

listing file

1 program test (input, output);

2 var a : integer;

3 var b : real;

4 var c : array [1..2] of integer;

5 var f : array [25..45] of real;

6 var g : array [1..2] of integer;

7 function fun1 : integer;

8 function fun5 : integer;

9 begin

10 end;

11 begin

12 end;

13 function fun2(x: integer; y: real) : real;

14 var e: real;

15 var ab : array [1..2500] of real;

16 var c: integer;

17 function fun3 : integer;

18 var l: integer;

19 begin

20 end;

21 function fun6 : integer;

22 begin

23 l:= 75

SEMERR: Identifier l is either not defined or not within scope

24 end;

25 function fun7 : integer;

26 var g: integer;

27 begin

28 end;

29 function fun4(hello: array [1..2] of integer) : integer;

30 begin

31 while(a=hello[2.5]) do

SEMERR: Identifiers cannot be indexed with a non-integer

SEMERR: invalid operands for operator =, operands must be of type boolean or operands must be of type integer/real

32 begin

33 e:=c

34 end

SEMERR: Type mismatch on assignment expecting type real received type integer

35 end;

36 begin

37 b := e + 4.44;

38 b:= (a mod x) / x;

39 while ((a >= 1) and ((b <= e) or (not (a = c)))) do

40 begin

41 c:= 2.5E2

42 end;

SEMERR: Type mismatch on assignment expecting type integer received type real

43 fun2 := 2.5

44 end;

45

46 begin

47 b:= fun2(2,f[5]);

48 if (a>2) then a:= 2 else a := a+2;

49 if (b > 4.2) then a := c[a]

50 end.

with semantic and syntax errors

program test (input, output);

var a : integer;

var b : real;

var c : array [1..2] of integer;

var f : array [25..45] of real;

var ggggggggggggg : array [1..2] of integer;

function fun2(x: integer; y: real) : real;

var e: real;

var ab : array [1..2500] of real;

var c: integer;

begin

b := e + 4.44;

b:= (a mod x) / x;

while ((a >= 1) and ((b <= e) or (not (a = c)))) do

begin

c:= 2.5E2

end;

fun2 := 2.55555555555555555555

end;

begin

b:= fun2(2.5,f[5]);

if (a>2) then a:= 2 else a := a+2;

if (b > 4.2) then a := c[a]

end.

Listing file

1 program test (input, output);

2 var a : integer;

3 var b : real;

4 var c : array [1..2] of integer;

5 var f : array [25..45] of real;

6 var ggggggggggggg : array [1..2] of integer;

LEXERR: Identifier is greater than 10 characters ggggggggggggg

SYNERR: token mismatch, expecting token Identifier instead received token ggggggggggggg

7 function fun2(x: integer; y: real) : real;

8 var e: real;

9 var ab : array [1..2500] of real;

10 var c: integer;

11 begin

12 b := e + 4.44;

13 b:= (a mod x) / x;

14 while ((a >= 1) and ((b <= e) or (not (a = c)))) do

15 begin

16 c:= 2.5E2

17 end;

SEMERR: Type mismatch on assignment expecting type integer received type real

18 fun2 := 2.55555555555555555555

LEXERR: Real number back is greater than 5 digits: 2.555555555...

SYNERR: tok mismatch expecting Identifier, Number, not, +, -, (, instead received 2.555555555...

19 end;

SEMERR: Type mismatch on assignment expecting type real received type type error

20

21 begin

22 b:= fun2(2.5,f[5]);

SEMERR: Function parameter type mismatch expecting integer, received real

SEMERR: Type mismatch on assignment expecting type real received type type error

23 if (a>2) then a:= 2 else a := a+2;

24 if (b > 4.2) then a := c[a]

25 end.

Address file

node name address scope function

a 0 test

b 4 test

c 12 test

f 20 test

e 196 fun2

ab 204 fun2

c 20204 fun2