Left Attributed Definitions

----------------------------------------

The beginning of a program

program → **program id (** identifier\_list **);** program1 {done}

{checkAddGreenNode(id.lexval, PNAME)}

program1 → declarations program1\_1

program1 → subprogram\_declarations compound\_statement **.**

program1 → compound\_statement **.**

program1\_1 → subprogram\_declarations compound\_statement **.**

program1\_1 → compound\_statement **.**

The list of program parameters

identifier\_list → **id** identifier\_list1 {done}

{checkAddBlueNode(id.lex, PPARAM)}

identifier\_list1 → **, id** identifier\_list1 {done}

{checkAddBlueNode(id.lex, PPARAM)}

identifier\_list1→ **epsilon**

The list of GLOBAL variables, which must be defined before subprograms (aka functions)

declarations → **var id :** type **;** declarations1 {done}

{checkAddBlueNode(id.lex, type.type)}

declarations1 → **var id :** type **;** declarations1 {done}

{checkAddBlueNode(id.lex, type.type)}

declarations1 → **epsilon**

The type of a variable, array, or an anything for that matter

type → standard\_type {done}

{type.type := standard\_type.type}

type → **array [ num .. num ] of** standard\_type {done}

{type.type := array(s, t)}

A subclass of type that only includes integers and reals. NOT arrays, although arrays must be an array of standard\_types in this grammar.

standard\_type → **integer** {done}

{standard\_type.type := intType}

standard\_type → **real** {done}

{standard\_type.type := realType}

This is the beginning of a subprogram inside a grammar. It comes after global variable declarations. Subprograms can be nested within other subprograms.

subprogram\_declarations → subprogram\_declaration **;** subprogram\_declarations1

subprogram\_declarations1 → subprogram\_declaration **;** subprogram\_declarations1 subprogram\_declarations1 → **epsilon**

subprogram\_declaration → subprogram\_head subprogram\_declaration1

subprogram\_declaration1 → declarations subprogram\_declaration1\_1

subprogram\_declaration1 → subprogram\_declarations compound\_statement

subprogram\_declaration1 → compound\_statement

subprogram\_declaration1\_1 → subprogram\_declarations compound\_statement subprogram\_declaration1\_1 → compound\_statement

The head of a function/subprogram, which includes the function name, parameters, and local variables for the scope.

subprogram\_head → **function id** subprogram\_head1 {done}

{checkAddGreenNode(id.lexval, subprogram\_head1.type)}

subprogram\_head1 → **(** parameter\_list **)** **:** standard\_type **;** {done}

{subprogram\_head1.type := standard\_type.type}

subprogram\_head1 → **:** standard\_type **;** {done}

{subprogram\_head1.type := standard\_type.type}

parameter\_list → **id** : type parameter\_list1 {done, but should test}

{checkAddBlueNode(id.lexval, type.type)} #must convert type value to function parameter equivalent type

parameter\_list1 → **; id** **:** type parameter\_list1

{checkAddBlueNode(id.lexval, type.type)} #must convert type value to function parameter equivalent type

parameter\_list1 → **epsilon**

This is a list of statements (bracketed by begin and end) that form a single unit. They can be found as a direct part of a program segment itself or inside a subprogram.

compound\_statement → **begin** compound\_statement1 {NOT done. Needs to return a type for statement}

compound\_statement1 → statement\_list **end**

compound\_statement1 → **end**

statement\_list → statement statement\_list1

statement\_list1 → **;** statement statement\_list1

statement\_list1 → **epsilon**

A single statement in a function. Delineated with ';'.

statement → variable **assignop** expression {done}

statement → compound\_statement {done}

statement → **if** expression **then** statement[1] statement1 {done}

statement → **while** expression **do** statement {done}

statement1 → **else** statement{done}

statement1 → **epsilon** {done}

The usage of a variable. id or id[exp]

variable → **id** variable1 {done}

{variable1.in := getType(id)}

{variable.type := getType(id)}

---begin subset---

variable1 → **[** expression **]** {done, but should test}

{ variable1.type := if expression.type = INT and variable1.in = array(s, t), then t

else ERR\* }

variable1 → **epsilon**

{ variable1.type := void}

expression\_list → expression expression\_list1 {done, but should test}

{expression\_list.type := if expression\_list1.type = s, then appendTo(expression\_list1.type, expression.type),

else ERR\*}

expression\_list1 → **,** expression expression\_list1 {done, but should test}

{expression\_list1.type := if expression\_list1.type = s, then appendTo(expression\_list1.type, expression.type),

else ERR\*}

expression\_list1 → **epsilon** {done, but should test}

{expression\_list1 := newS()} #an empty list for argument types

expression → simple\_expression expression1

{ expression1.in := simple\_expression.type }

{ expression.type := simple\_expression.type }

expression1 →**relop** simple\_expression

{ expression1.type := if expression1.in = simple\_expression.type = (int or real) then BOOL

else ERR\*}

expression1 → **epsilon**

{expression1.type := void}

simple\_expression → term simple\_expression1

{simple\_expression.type := if term.type = simple\_expression1.type or simple\_expression1 := void then term.type

else ERR\*}

simple\_expression → sign term simple\_expression1

{simple\_expression.type := if term.type = simple\_expression.type or simple\_expression1 := void then term.type

else ERR\*}

simple\_expression1 → **addop** term simple\_expression1[1]

{simple\_expression1.type := if term.type = simple\_expression1[1].type or simple\_expression1[1].type = void then term.type

else ERR\* }

simple\_expression1 → **epsilon**

{ simple\_epression1.type := void }

term → factor term1

{term.type := if factor.type = term1.type or term1.type = void then factor.type

else ERR\* }

term1 → **multop** factor term1[1]

{term1.type := if (factor.type = term1[1].type = (int or real)) or term1[1].type = void then factor.type

else ERR\* }

term1 → **epsilon**

{ term1.type := void}

factor → **id** factor1{not done: function calls}

{factor.type := if getType(id) = tArray and factor1.type = tArray, then tArray

else if getType(id) = FNAME and factor1.type = AN ARRAY DATA STRUCTURE!, then factor1.type

else if getType(id) = (int or real) and factor1.type = void, then getType(id)

else ERR\*}

{factor1.in := getType(id)}

factor → **num**

{factor.type := num.type}

factor → **(** expression **)**

{factor.type:= expression.type}

factor → **not** factor[1]

{ factor.type := if factor[1].type = BOOL then BOOL

else if factor[1] = ERR or ERR\* then ERR

else ERR\* }

factor1 → **(** expression\_list **)** {done...ish. Should definitely test}

{ factor1.type := if expression\_list.type = s then s,

else ERR\*} #s is a list of the types of all the expressions

factor1 → variable1

{factor1.type := varable1.type} #the type will be array or void

{variable1.in := factor1.in.type}

sign → **+**

sign → **-**