Grammar with Decorations

Return type (s)

Global
Synthesized
Inherited (or other semantic action)

bool — true on success, false otherwise

checkAddGreenNode(id.lex, PGNAME)
offset = 0

1.	void	*program*	→	**program id** (*identifier_list*) ; *declarations* *subprogram_declarations* *compound_statement* .
2.1.	void	*identifier_list*	→	**id** *identifier_list'*
2.2.1.	void	*identifier_list'*	→	, **id** *identifier_list'*
2.2.2.		*identifier_list'*	→	ϵ
3.1.	void	*declarations*	→	**var id** : *type* ; *declarations*
3.2.		*declarations*	→	ϵ
4.1.	TYPE	*type*	→	*standard_type*
4.2.		*type*	→	**array** [**num** .. **num**] **of** *standard_type*
5.1.	TYPE	*standard_type*	→	**integer**
5.2.		*standard_type*	→	**real**
6.1.	void	*subprogram_declarations*	→	*subprogram_declaration* ; *subprogram_declarations*
6.2.		*subprogram_declarations*	→	ϵ
7.	void	*subprogram_declaration*	→	*subprogram_head declarations* *subprogram_declarations compound_statement*
8.	void	*subprogram_head*	→	**procedure id** *arguments* ;
9.1.	void	*arguments*	→	(*parameter_list*)
9.2.		*arguments*	→	ϵ
10.1.	void	*parameter_list*	→	**id** : *type parameter_list'*
10.2.1.	void	*parameter_list'*	→	; **id** : *type parameter_list'*
10.2.2.		*parameter_list'*	→	ϵ
11.	void	*compound_statement*	→	**begin** *optional_statements* **end**
12.1.	void	*optional_statements*	→	*statement_list*
12.2.		*optional_statements*	→	ϵ
13.1.	void	*statement_list*	→	*statement statement_list'*
13.2.1.	void	*statement_list'*	→	; *statement statement_list'*
13.2.2.		*statement_list'*	→	ϵ
14.1.	void	*statement*	→	*variable* **assignop** *expression*
14.2.		*statement*	→	*procedure_statement*
14.3.		*statement*	→	*compound_statement*
14.4.		*statement*	→	**while** *expression* **do** *statement*
14.5.		*statement*	→	**if** *expression* **then** *statement else'*
15.1.	void	*else'*	→	**else** *statement*
15.2.		*else'*	→	ϵ
16.	TYPE	*variable*	→	**id** *array_access*
17.1.	TYPE	*array_access*	→	[*expression*]
17.2.		*array_access*	→	ϵ
18.	void	*procedure_statement*	→	**call id** *optional_expressions*
19.1.	void	*optional_expressions*	→	(*expression_list*)
19.2.		*optional_expressions*	→	ϵ
20.1.	void	*expression_list*	→	*expression expression_list'*
20.2.1.	void	*expression_list'*	→	, *expression expression_list'*
20.2.2.		*expression_list'*	→	ϵ
21.	TYPE	*expression*	→	*simple_expression related_expression*
22.1.	TYPE	*related_expression*	→	**relop** *simple_expression*
22.2.		*related_expression*	→	ϵ
23.1.1.	TYPE	*simple_expression*	→	*term simple_expression'*
23.1.2.		*simple_expression*	→	*sign term simple_expression'*
23.2.1.	TYPE	*simple_expression'*	→	**addop** *term simple_expression'*
23.2.2.		*simple_expression'*	→	ϵ

checkAddBlueNode(id.lex, PGNAME)

checkAddBlueNode(id.lex, type.type)

type ← stdtype.type width ← st.width

type ← AINT if INT, AREAL if REAL
width ← (num.num + 1) * st.width if ERR, ERR* otherwise
offset += width

type ← INT width ← 4
type ← REAL width ← 8

popFromGreenStack

checkAddGreenNode(id.lex, PROC)
offset = 0

checkAddBlueNode(id.lex, "pp" ++ type)

s.t	v.t	e.t
ERR*	UNDECL	—
—	INT	INT
—	REAL	REAL
—	ERR	—
ERR*	Anything	Anything

Check type BOOL

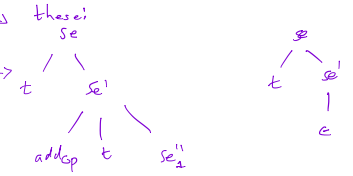
Same as factor → id factor'

Must exist & be a procedure

i ← pointer to the first item

i ← pointer to the right type - if match, continue; else, fail

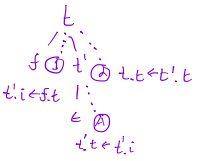
Looks exactly like the table for * and / for factor; so do these!

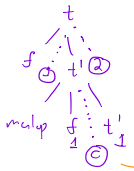


1

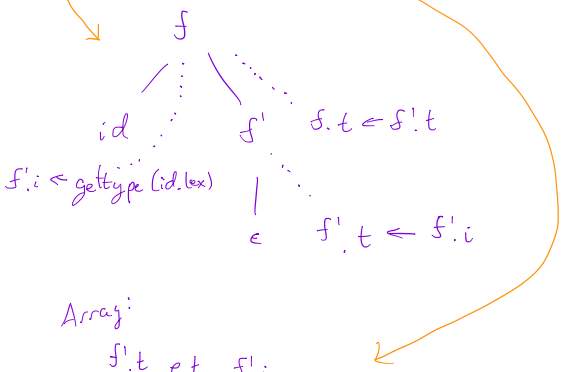
or

24.1.	TYPE	*term*	→	*factor term′*
24.2.1.	TYPE	*term′*	→	**mulop** *factor term′*
24.2.2.		*term′*	→	**ϵ**
25.1.1.	TYPE	*factor*	→	**id** *factor'* {type ← factor'.type}
25.1.2.		*factor*	→	**num** {type ← num.type}
25.1.3.		*factor*	→	(*expression*) {type ← exp.type}
25.1.4.		*factor*	→	**not** *factor*
25.2.1.		*factor'*	→	[*expression*] See table down below
25.2.1.		*factor'*	→	**ϵ** {type ← gettype(id.lex)} (undeclared error, procedure, etc. - use a table)
26.1.	void	*sign*	→	+
26.2.		*sign*	→	−





f.type	f1.type
BOOL	BOOL
ERR*	¬BOOL ^ ¬ERR
ERR	ERR



Array:

f'.t	e.t	f'.i
INT	INT	AINT
REAL	INT	AREAL
ERR*	¬INT ^ ¬ERR	
ERR*		ARRAY
ERR	ERR	
ERR		ERR

AINT :: Array of ints
AREAL :: Array of reals

Mulops:
*
/
div
mod
and

(* or /)

t.i	f.t	mulop.op	t'.i
INT	INT	*	INT
REAL	REAL	*	REAL
ERR	ERR	*	ERR
ERR*	INT	*	REAL

But there is no type coercion

Watch out for the possibility of array types!!