An introduction to TACL

Informal and incomplete

Declarations

```
var type name = initialisation ;
proc name ( formal-arguments ) [ body ]
fun type name ( formal-arguments ) = return-value ;
fun type name ( formal-arguments ) [ body ^ return-value ]
```

Statements

```
variable = expression ;
name ( actual-arguments ) ;
print expression ;
if ( condition ) statement
if ( condition ) statement else statement
while ( condition ) statement
[ statements ]
```

Basics in compiling

TACL code

```
# returns twice n
fun int twice(int n) = 2 * n;
```

Token sequence

FUN INT ID(twice) OPPAR INT ID(n) CLPAR EQSIGN
INT_LITERAL(2) TIMES ID(n) SEMICOLON

Grammar (partial)

$$
\begin{aligned}
\langle \text{fundef} \rangle &\rightarrow \text{FUN } \langle \text{type} \rangle \text{ ID OPPAR } \langle \text{formal-args} \rangle \text{ CLPAR} \\
&\qquad \langle \text{body} \rangle \\
\langle \text{type} \rangle &\rightarrow \text{INT} \mid \text{REAL} \mid \text{BOOL} \\
\langle \text{formal-arg} \rangle &\rightarrow \langle \text{type} \rangle \text{ ID} \\
\langle \text{formal-args} \rangle &\rightarrow \langle \text{formal-arg} \rangle \ \langle \text{more-formal-args} \rangle \mid \lambda \\
\langle \text{more-formal-args} \rangle &\rightarrow \text{COMMA } \langle \text{formal-arg} \rangle \ \langle \text{more-formal-args} \rangle \mid \lambda \\
\langle \text{body} \rangle &\rightarrow \text{EQSIGN } \langle \text{expression} \rangle \text{ SEMICOLON} \\
\langle \text{expression} \rangle &\rightarrow \langle \text{expression} \rangle \text{ TIMES } \langle \text{expression} \rangle \\
&\quad \mid \ \langle \text{atomic-expression} \rangle \\
\langle \text{atomic-expression} \rangle &\rightarrow \text{ID} \mid \langle \text{literal} \rangle \\
\langle \text{literal} \rangle &\rightarrow \text{INT\_LITERAL}
\end{aligned}
$$

Lexical analysis

Specifying a tokeniser

	flex	JFlex
`fun`	`return FUN;`	`{ return token(sym.FUN); }`
`int`	`return INT;`	`{ return token(sym.INT); }`
`=`	`return EQSIGN;`	`{ return token(sym.EQSIGN); }`
`;`	`return SEMICOLON;`	`{ return token(sym.SEMICOLON); }`
`"("`	`return OPPAR;`	`{ return token(sym.OPPAR); }`
`")"`	`return CLPAR;`	`{ return token(sym.CLPAR); }`
`\*`	`return TIMES;`	`{ return token(sym.TIMES); }`
`[0-9]+`	`return INT_LITERAL;`	`{ return token(sym.INT_LITERAL); }`

Note: token attributes are not considered here

Syntactic analysis

Writing the grammar

Bison

```
fundef : FUN type ID OPPAR formal_args CLPAR body ;
type : INT | REAL | BOOL ;
formal_arg : type ID ;
formal_args : formal_arg more_formal_args | ;
more_formal_args : COMMA formal_arg more_formal_args | ;
body : EQSIGN expression SEMICOLON ;
expression : expression TIMES expression | atomic_expression ;
atomic_expression : ID | literal ;
literal : INT_LITERAL ;
```

CUP

```
fundef ::= FUN type ID OPPAR formal_args CLPAR body ;
type ::= INT | REAL | BOOL ;
formal_arg ::= type ID ;
formal_args ::= formal_arg more_formal_args | ;
```

Etc.

Common patterns in grammars

Iteration (a.k.a. repetition)

$$X \rightarrow \ldots \text{ something}$$

A non-empty sequence of Xs

$$Xs_+ \rightarrow X \mid X\ Xs_+$$

A (possibly empty) sequence of Xs

$$Xs_* \rightarrow \lambda \mid X\ Xs_*$$

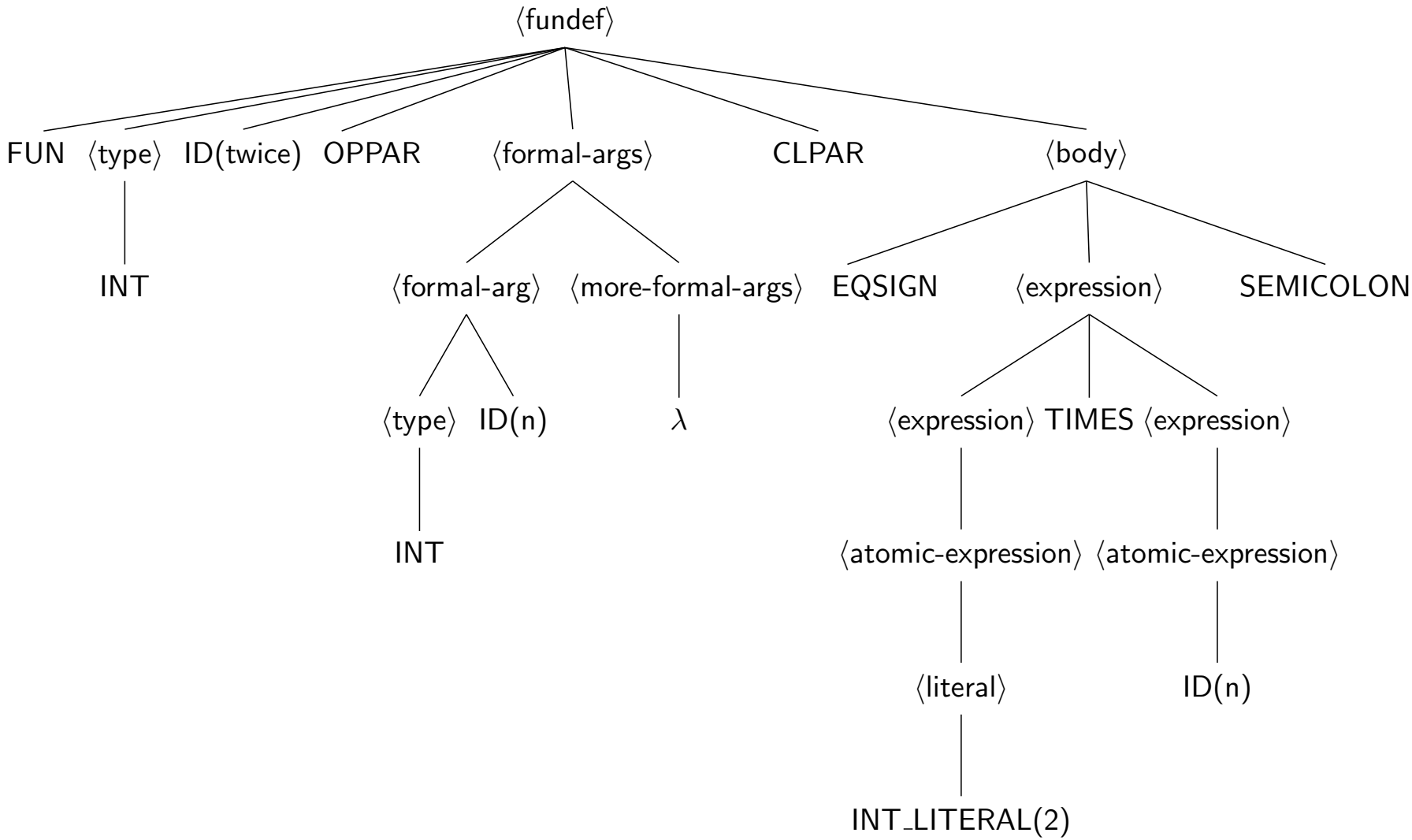
A (non-empty) separated sequence of Xs

$$Xs_{s+} \rightarrow X \mid X \text{ separator } Xs_{s+}$$

A possibly empty separated sequence of Xs

$$Xs_{s*} \rightarrow \lambda \mid Xs_{s+}$$

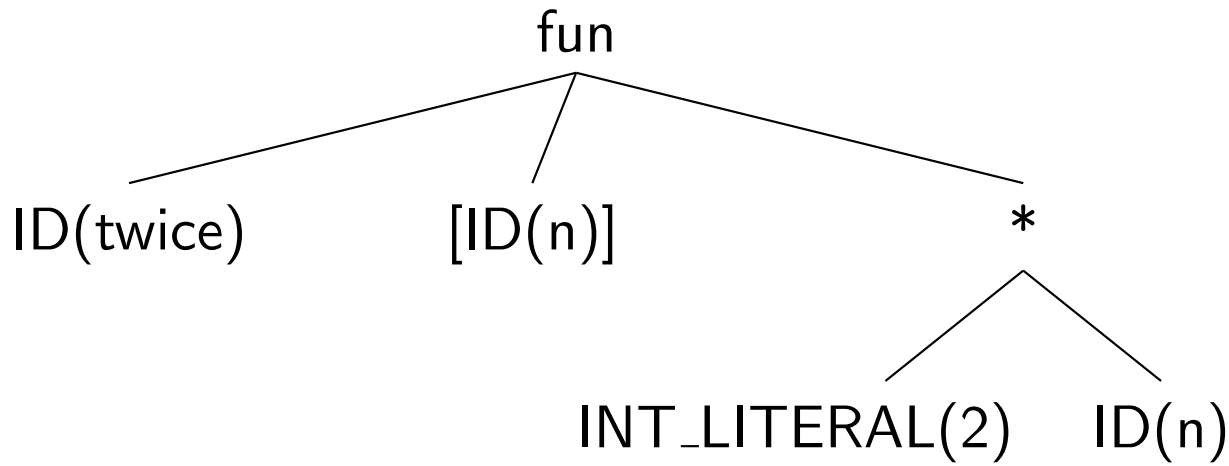
Parse tree



Too verbose. . .

Abstract syntax tree (AST)

Extreme version



Where have all the types gone. . . ? Stay tuned

Abstract syntax

⟨fundef⟩	→ ⟨type⟩ ID ⟨formal-args⟩ ⟨body⟩	(fun)
⟨type⟩	→ INT | REAL | BOOL	*(int | real | bool)*
⟨formal-arg⟩	→ ⟨type⟩ ID	*(arg)*
⟨formal-args⟩	→ ⟨formal-arg⟩ ⟨more-formal-args⟩ | λ	(...)
⟨more-formal-args⟩	→ ⟨formal-arg⟩ ⟨more-formal-args⟩ | λ	(...)
⟨body⟩	→ ⟨expression⟩	
⟨expression⟩	→ ⟨expression⟩ ⟨expression⟩	(times)
	| ⟨atomic-expression⟩	
⟨atomic-expression⟩	→ ID	(id)
	| ⟨literal⟩	
⟨literal⟩	→ INT_LITERAL	(int_literal)

Example program

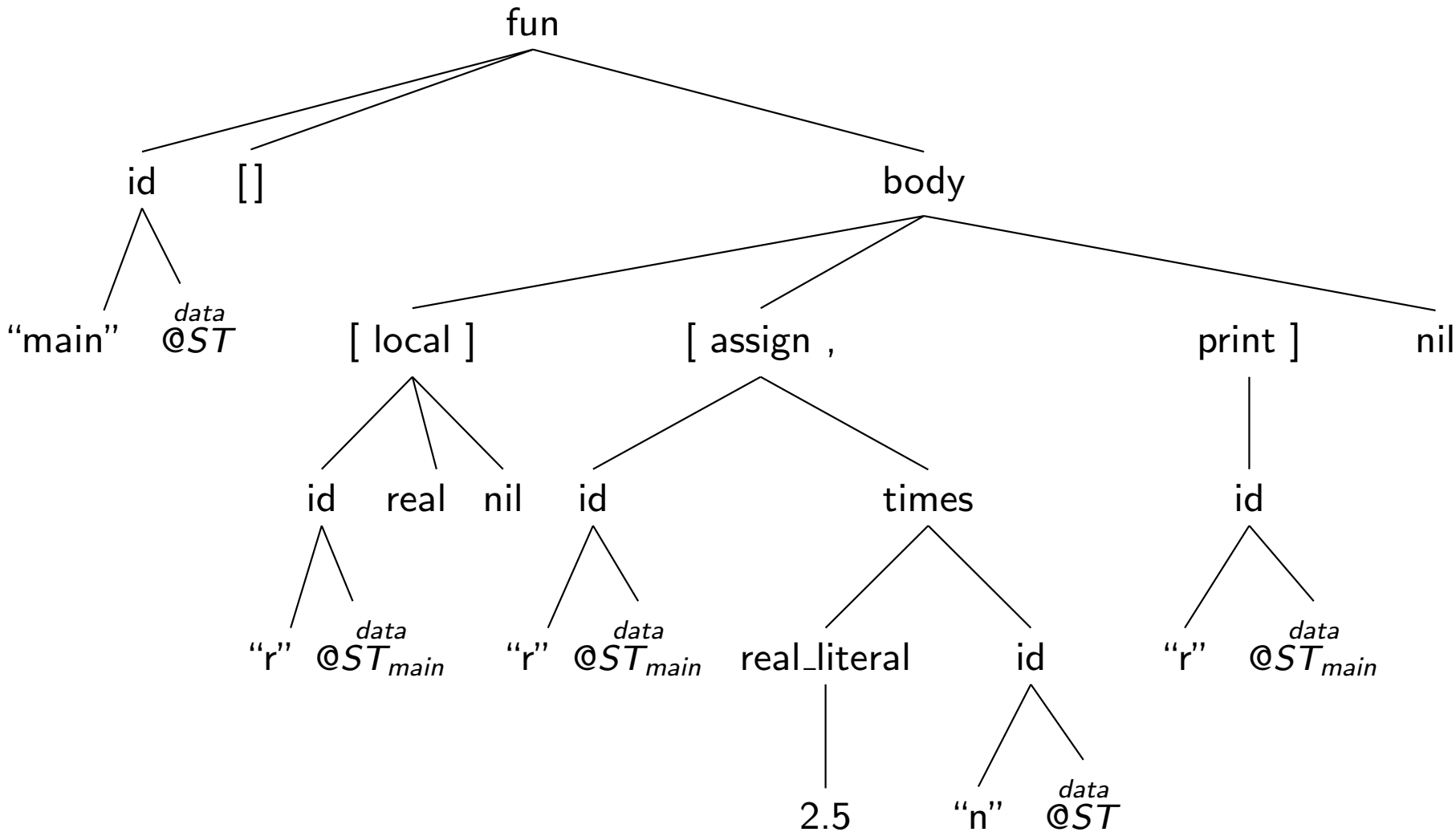
```
var int n = 3;

proc main()
[
  var real r;

  r = 2.5 * n;

  print r;
]
```

AST for `main`


fun
id
[]
body
"main"
data
@ST
[local]
[assign ,
print]
nil
id
real
nil
id
times
id
"r"
data
@ST_main
"r"
data
@ST_main
real_literal
id
"r"
data
@ST_main
2.5
"n"
data
@ST

Program symbol table

Global symbol table (ST)

name	*kind*	*type*	...
"n"	var	int	
"main"	fun	void	locals = ST_{main}

`main` symbol table (ST_{main})

name	*kind*	*type*	...
"r"	local	real	

Type checking

Type system

Axioms

$$\frac{}{(\text{int\_literal } n) : \text{int}} \qquad \frac{}{\text{false} : \text{bool}}$$

Rules

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{(\text{times } e_1\ e_2) : \text{int}} \qquad \frac{e_1 : \text{real} \quad e_2 : \text{real}}{(\text{times } e_1\ e_2) : \text{real}}$$

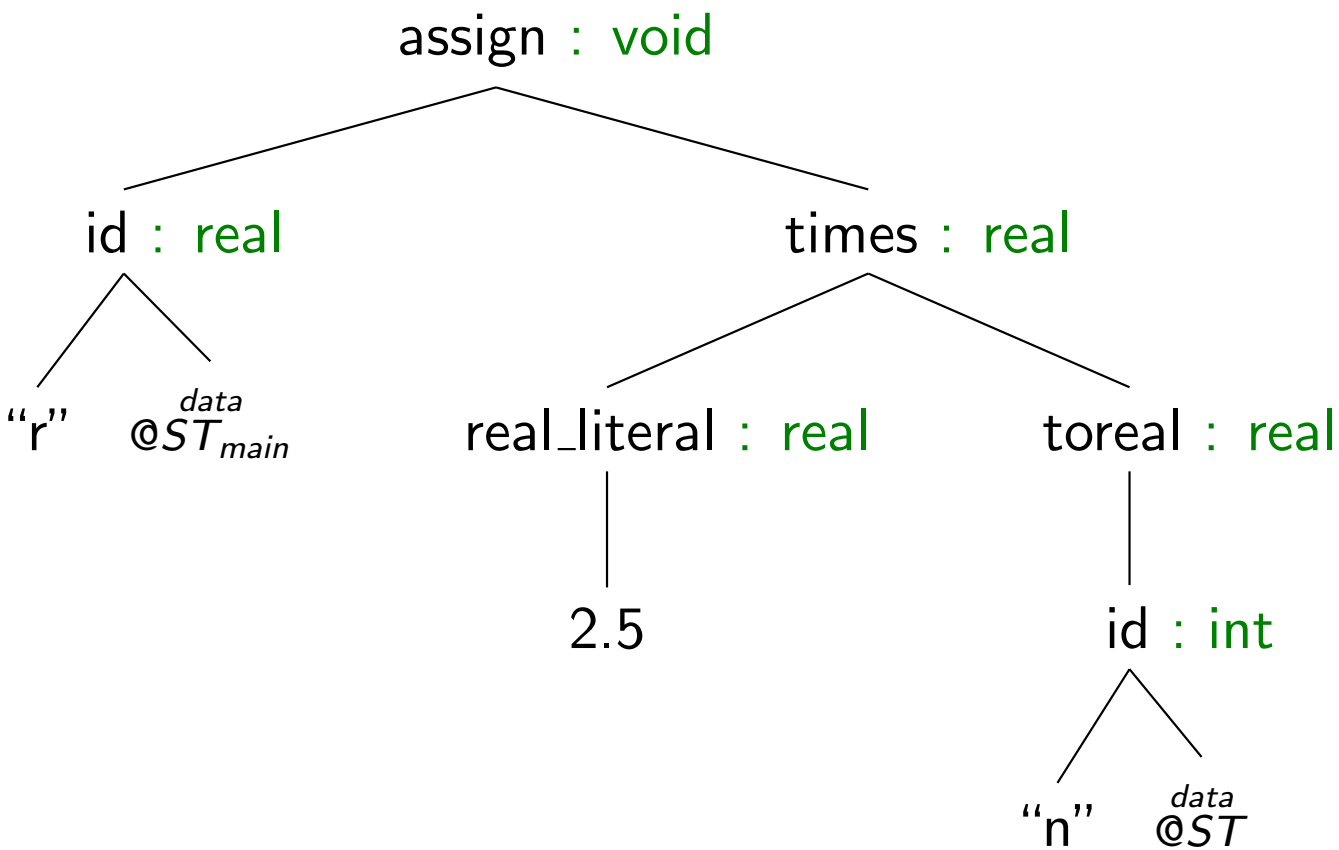
$$\frac{e_1 : \tau \quad \tau \neq \text{int} \quad \tau \neq \text{real}}{(\text{times } e_1\ e_2) : \text{error}} \qquad \frac{e_2 : \tau \quad \tau \neq \text{int} \quad \tau \neq \text{real}}{(\text{times } e_1\ e_2) : \text{error}}$$

$$\frac{e_1 : \text{real} \quad e_2 : \text{int} \quad e_2' = (\text{toreal } e_2) : \text{real}}{(\text{times } e_1\ e_2) \mapsto (\text{times } e_1\ e_2') : \text{real}}$$

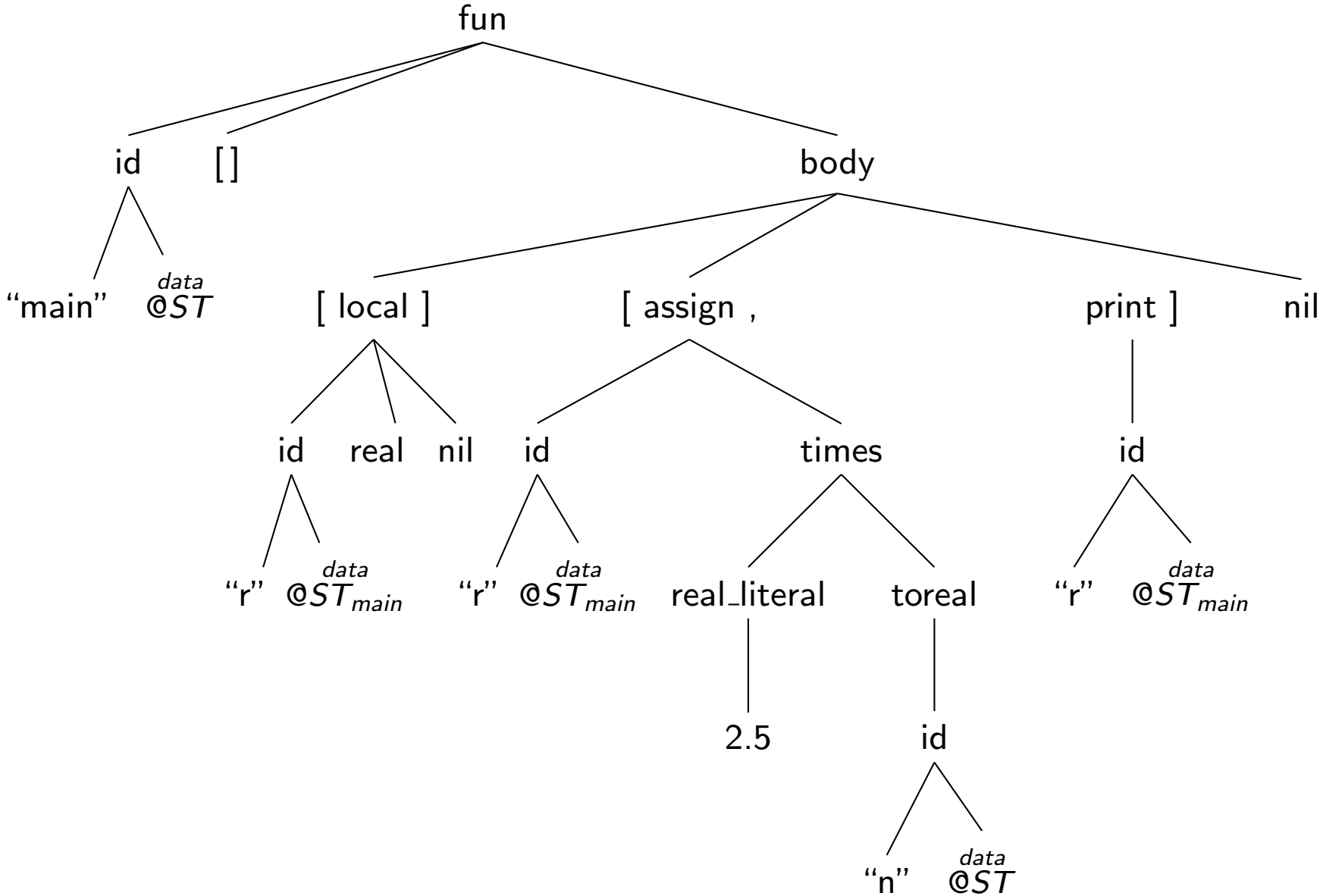
$$\frac{e_1 : \text{int} \quad e_2 : \text{real} \quad e_1' = (\text{toreal } e_1) : \text{real}}{(\text{times } e_1\ e_2) \mapsto (\text{times } e_1'\ e_2) : \text{real}}$$

Rewriting the AST

The new assign subtree



AST for `main` after type checking



Note: type annotations have been omitted due to space considerations