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)

⟨fundef⟩	→ FUN ⟨type⟩ ID OPPAR ⟨formal-args⟩ CLPAR ⟨body⟩
⟨type⟩	→ INT | REAL | BOOL
⟨formal-arg⟩	→ ⟨type⟩ ID
⟨formal-args⟩	→ ⟨formal-arg⟩ ⟨more-formal-args⟩ | λ
⟨more-formal-args⟩	→ COMMA ⟨formal-arg⟩ ⟨more-formal-args⟩ | λ
⟨body⟩	→ EQSIGN ⟨expression⟩ SEMICOLON
⟨expression⟩	→ ⟨expression⟩ TIMES ⟨expression⟩
	| ⟨atomic-expression⟩
⟨atomic-expression⟩	→ ID | ⟨literal⟩
⟨literal⟩	→ INT_LITERAL

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)

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

A non-empty sequence of Xs

$$Xs_+ \to X \mid X \, Xs_+$$

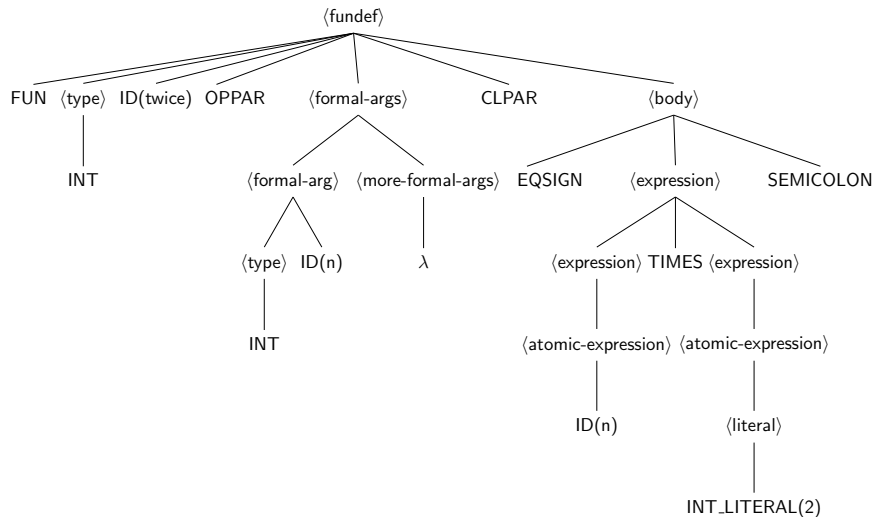A (possibly empty) sequence of Xs

$$Xs_* \to \lambda \mid X \, Xs_*$$

A (non-empty) separated sequence of Xs

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

A possibly empty separated sequence of Xs

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

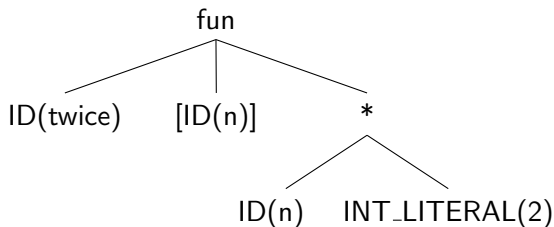# Parse tree

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)

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 ]
```
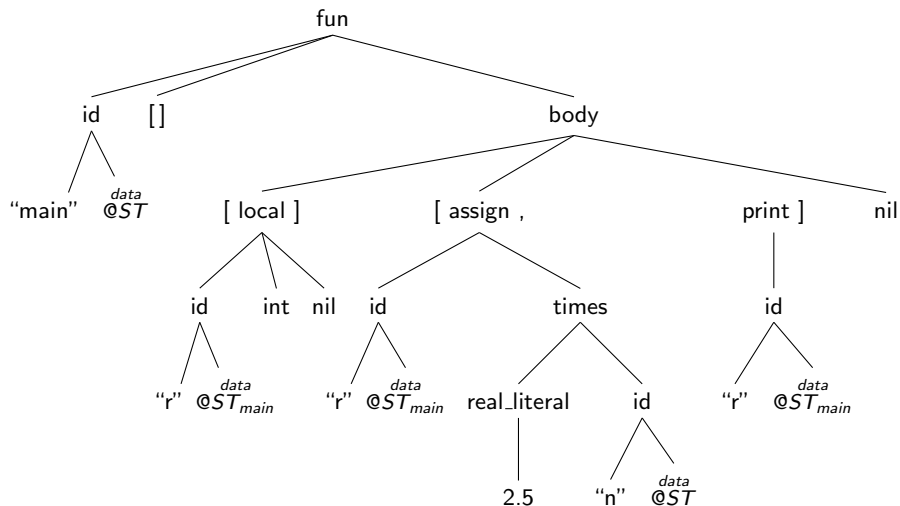
Example program

```
var int n = 3;

proc main()
[
  var real r;

  r = 2.5 * n;

  print r;
]
```

AST for `main`

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	