Programming Languages — Subprograms, AST

Subprograms

- **Signature** *name* plus order and type of parameters
- **Types** *function* (returns a value); *procedure* (`void` or no return value)
- **generic** variable on some type in the signature
- **overloaded** reuse of subprogram name with different *signature*
- **override** reuse of a full *signature* in a subclass

The 5 Properties

1. Named — A subprogram has a name
   Relaxed: unnamed subprograms; `lambda`
2. Explicitly called — Control is transferred to the subprogram through a line calling it
   Relaxed: exception handler (or any other handler); callback
3. Immediate transfer of control — Called subprogram begins as soon as call happens
   Relaxed: deferred execution
4. Caller suspended — Calling subprogram stops making progress until callee returns control and only then continues after call site
   Relaxed: concurrency, multiple threads of control
5. Callee finishes, returning control to caller — Called subprogram runs to completion, setting return value if necessary, and returns control to site of call; callee context is no longer available
   Relaxed: coroutines, `yield`, generators, lazy lists

Parameters

Formal Listed when subprogram is *defined*; in scope within the subprogram.
Actual Provided at the call to the subprogram; evaluated in calling scope, assigned to formal parameters. Method of assignment depends on parameter passing *semantics*

value actual parameter evaluated and value put in new, local variable for the formal parameter.
(pointer) programmer takes *address* of actual parameter and that is passed by value to the formal parameter; must be manually dereferenced.
reference `lvalue` of actual parameter determined and set as `lvalue` of formal parameter.
copy also **value-return**: parameter passed in by *value*; at end of called subprogram, value of formal parameter copied back into actual parameter.
name the *text* of the actual parameter is passed into the callee and evaluated in the callee's scope when the formal parameter is used. This is a lot like macro expansion.

Pass by Name

```
function returnTwice(byName : integer): integer;
begin
  returnTwice := byName + byName
end;
```

If we call with *actual parameter* that has no side effects, all is well:

```
i := 5;
j := returnTwice(i);
writeln(i, j);
```

Gives us 5 and 10, as expected.
Call with an actual parameter with a *side effect*:

```
i := 5;
j := returnTwice(inc(i));
writeln(i, j);
```

and while we might expect 6 (5 incremented) and 12, we instead get 7 (because increment was called once *each time the parameter was evaluated*) and 13 (first increment returned 6, the second 7; do the math). So `returnTwice`, which looks like it would return double its parameter (and therefore a multiple of two), returns an *odd number*.

Parsing, AST

Language processing is going from a *stream of characters* to a *stream of instructions*.
In lexical analysis, the stream of characters is transformed into a stream of tokens. That stream of tokens is then *parsed* into an **abstract syntax tree** (AST) (or equivalent).
there must be some intermediate form that represents the **meaning** of the program so that the instructions can preserve that meaning.
Grammar

```
<expression> ::= ( + <expression> <term> ) | ( - <expression> <term> ) | <term>
<term>       ::= ( * <term> <factor> ) | ( / <term> <factor> ) | <factor>
<factor>     ::= <expression> | [int] | [var]
```

```
(* x (+ 2 y))
```