Utrecht University

Concepts of Programming Language Design

TinyC

Gabriele Keller
Tom Smeding

TinyC : The essence of imperative and procedural programming

- What is imperative/ procedural programming?
  - declarative (functional and logic) languages describe ***what*** needs to be done, imperative languages ***how*** it needs to be done.
  - strictly speaking, procedural programming is imperative programming with subroutines/procedures (often used is synonymously)
  - closer to machine language, where the program is also expressed in terms of commands:
    - store/update value x at location y
    - add values stored in a certain location
  - more abstract than machine language:
    - symbolic names, subroutines, loop constructs

The evolution of imperative and procedural programming languages

- Assembly languages (around the 1950's)
  - a more human-readable representation of machine code instructions
  - mnemonic codes to represent machine language instructions
  - early examples:
    - Assembly for UNIVAC I and Assembly for IBM 701.
  - more recently
    - 70's: for Intel's 8080 and x86 architecture
    - 80's: ARM
- Still used
  - device drivers, micro controllers
  - real time systems
  - parts of an operating system (boot loaders)
  - firmware

Utrecht University

The evolution of imperative and procedural programming languages

- Fortran (1957):
  - (**For**mula **Tran**slation)
  - developed by IBM (John W. Backus was one of the designers)
  - one of the earliest high-level programming languages.
  - designed for scientific and engineering calculations
  - introduced the concept of procedural programming

```
! Procedure to sum the elements of an array
function sumArray(arr, size) result(sum)
  integer, intent(in) :: arr(:)
  integer, intent(in) :: size
  integer :: sum
  integer :: i

  sum = 0
  do i = 1, size
    sum = sum + arr(i)
  end do
end function sumArray
```

The evolution of imperative and procedural programming languages

- Algol (1958):
  - **Algo**rithmic **L**anguage
  - developed by a committee of European and American computer scientists (Backus, Bauer, Green, Katz, McCarthy, Naur, Perlis, Rutishauser, Samelson, van Wijngaarden, Vauquois, Wegstein, Woodger)
  - influential role in the development of programming languages
  - introduced the concept of block structures.

```
PROCEDURE sumArray(arr, size)
    VALUE arr, size; INTEGER arr[size], size;
    INTEGER i, sum

    sum := 0
    FOR i := 1 STEP 1 UNTIL size DO
        sum := sum + arr[i]

    RETURN sum
```

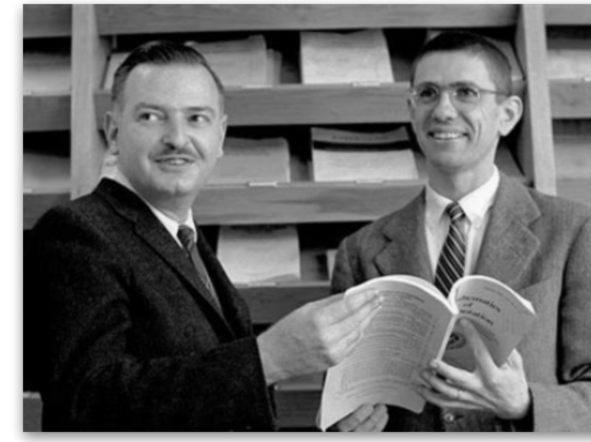
The evolution of imperative and procedural programming languages

- Cobol (1959):
  - **Co**mmon **B**usiness-**O**riented **L**anguage
  - developed by Grace Hopper
  - for business, finance, admin tasks
  - focus on readability
  - designed to be easy for 'non-programmers'

```
PROCEDURE DIVISION.
   PERFORM VARYING array-element FROM 1 BY 1
     UNTIL array-element > 5
       ADD array-element TO totalSum
   END-PERFORM.

   DISPLAY 'The sum of the array elements is: ' totalSum.
```

The evolution of imperative and procedural programming languages



- BASIC
  - **B**eginner's **a**ll-purpose **s**ymbolic **i**nstruction **c**ode, by
  - John G. Kemeny, Thomas E. Kurtz, 1963
  - is a family of programming languages
  - BASIC was designed to provide an easy entry point to computer programming for non-specialists
    - Microsoft BASIC
    - Commodore BASIC
    - Atari BASIC
    - Sinclair BASIC

```
10 REM Procedure to sum the elements of an array
20 PROCEDURE sumArray(arr(), size, sum)
30   DIM arr(5)
40   LET sum = 0
50   FOR i = 1 TO size
60     LET sum = sum + arr(i)
70   NEXT i
80   RETURN
90 END PROCEDURE
```

The evolution of imperative and procedural programming languages

- Pascal (1970):
  - Designed by Niklaus Wirth, Pascal was created for teaching structured programming.
  - It emphasised clarity and good programming practices
  - contributing to the development of modern programming methodologies.

```
procedure SumArray(arr: IntArray;
                   size: Integer; var sum: Integer);
var
  i: Integer;
begin
  sum := 0;
  for i := 1 to size do
    sum := sum + arr[i];
end;
```

The evolution of imperative and procedural programming languages

- C (1972):
  - Successor of B (BCPL - **B**asic **C**ombined **P**rogramming **L**anguage).
  - Developed by Dennis Ritchie at Bell Labs, C became a widely used procedural programming language.
  - C influenced many later languages
  - Development language of the Unix operating system.

Utrecht University

TinyC : The essence of imperative and procedural programming

- Historically, not strong on abstraction
  - more modern imperative languages included support for modularisation
  - trend towards object oriented programming (e.g., C to C++) with the intention of improving maintainability
- Developments generally
  - towards structured control flow (loops etc) away from arbitrary jumps/gotos
  - more control of the name space (modules with control of visibility vs inclusion of full files)

TinyC : The essence of imperative programming

- An imperative language with the following features:
  - Global function (procedure) declarations
  - Global and local variables
  - Assignment
  - Iteration: `while`-loops
  - Conditional: `if`-statements
  - Only a single value type: `int`
  - For now, no pointers or the like
- Static semantics: how to model global and local declarations, blocks
- Dynamic semantics: side effects, non-local control flow (return statements)

Utrecht University

TinyC : The essence of imperative programming

- BNF:

$$
\begin{array}{lcl}
\mathit{prgm} & ::= & \mathit{gdecs}\ \mathit{stmt} \\
\mathit{gdecs} & ::= & \epsilon \mid \mathit{gdec}\ \mathit{gdecs} \\
\mathit{gdec} & ::= & \mathit{fdec} \mid \mathit{vdec} \\
\mathit{vdecs} & ::= & \epsilon \mid \mathit{vdec}\ \mathit{vdecs} \\
\mathit{vdec} & ::= & \texttt{int}\ \mathit{Ident} = \mathit{Int}; \\
\mathit{fdec} & ::= & \texttt{int}\ \mathit{Ident}\ (\mathit{arguments})\ \mathit{stmt} \\
\mathit{stmt} & ::= & \mathit{expr}; \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}; \mid \texttt{return}\ \mathit{expr}; \mid \\
 & & \{\ \mathit{vdecs}\ \mathit{stmts}\ \} \mid \texttt{while}\ (\ \mathit{expr}\ )\ \mathit{stmt} \\
\mathit{stmts} & ::= & \epsilon \mid \mathit{stmt}\ \mathit{stmts} \\
\mathit{expr} & ::= & \mathit{Int} \mid \mathit{Ident} \mid \mathit{expr} + \mathit{expr} \mid \mathit{expr} - \mathit{expr} \mid \\
 & & \mathit{Ident} = \mathit{expr} \mid \mathit{Ident}\ (\mathit{exprs}) \\
\mathit{arguments} & ::= & \epsilon \mid \texttt{int}\ \mathit{Ident}\ ,\ \mathit{arguments} \\
\mathit{exprs} & ::= & \epsilon \mid \mathit{expr}\ ,\ \mathit{exprs}
\end{array}
$$

Concrete Syntax

- Programs consist of
  - a sequence of global variable and function declarations and
  - a statement

$$prgm \quad ::= \quad gdecs\ stmt$$

- Function declarations

$$fdec \quad ::= \quad \texttt{int}\ Ident\ (\,arguments\,)\ stmt$$

```
int f (int x1, int x2, ...) stmt
```

- Variable declarations

$$vdec \quad ::= \quad \texttt{int}\ Ident = Int;$$

```
int x = v;
```

Utrecht University

Concrete Syntax

- Statements versus expressions
  - statements are mainly about effects (but also have values)
    - $expr$;
    - `if` $expr$ `then` $stmt$ `else` $stmt$
    - `while (`$expr$`)` $stmt$
    - `return (`$expr$`);`
    - `{`$ldec\ stmts$`}`: a block - local variable declarations followed by a sequence of statements
  - expressions are about values (but can also have effects)
    - arithmetic expressions
    - assignments: x `=` $expr$
    - function calls: f `(`$expr_1$`,` $expr_2$`, ... ,` $expr_n$`)`
    - variables, integer values

Concrete Syntax

- Variables
  - have to be initialised
  - not true variables in the mathematical sense (MinHs's are), but refer to the (changeable) content of a (fixed) memory location
  - we assume that variable names are not re-used their scope
  - Example program:

```
int result = 0;
int div (int x, int y) {
  int res = 0;
  while (x > y) {
    x = x - y;
    res = res + 1;
  }
  return res;
}

result = div (16, 5);
```

Abstract Syntax

- We skip this step - we know how to do it
- We continue with the concrete syntax (readability!)
- For the abstract syntax for TinyC, we would use first order abstract syntax, since the variables in TinyC do not behave like mathematical variables (so substitution is not allowed)

Static Semantics

- What kind of properties do we have to check?
  - Are all variables and functions declared before use?
  - Are functions called with the correct number of arguments?
  - What about `return` statements in functions?
    - could we check that every possible control flow in a function block ends with a `return` statement?
    - for now, we set the return value to the value of the last expression in the block in case there is no explicit `return` statement
- What kind of structure do we need to maintain for these checks?
  - Environment of variables: $V = \{x_1,\ x_2, \ldots\}$
  - Environment of functions with their arity: $F = \{f_1 : n_1,\ f_2 : n_2, \ldots\}$

Utrecht University

Static Semantics

- Two kinds of language components:
  - ⋆ expressions and statements
  - ⋆ declarations
- Two kinds of judgements:
  - well-formed expressions and judgements (given both a variable environment V and function environment F):
    - $V, F \vdash expr\ ok$

      given environments V and F, expr /stmt is well formed
    - $V, F \vdash stmt\ ok$
  - well-formed declarations (determining a variable and function environment)
    - $\vdash gdecs\ decs\ V, F$

      the global declarations gdecs are well formed and declare the environments V and F
    - $V, F \vdash ldecs\ decs\ V'$

      given V and F, the local decl. ldecs are well formed and declare the new environment V'
  - Note: we write $\vdash expr\ ok$ instead of $\varnothing, \varnothing \vdash expr\ ok$

Static Semantics

- Given a program

  $$\boldsymbol{gdecs\ \ stmt}$$

  we have

  $$\frac{\vdash\ \boldsymbol{gdecs\ \ decs\ \ V,\ \ F} \qquad \boldsymbol{V,F} \vdash\ \boldsymbol{stmt\ \ ok}}{\vdash\ \boldsymbol{gdecs\ \ stmt\ ok}}$$

That is, the program is well-formed if

- its global declarations are well-formed and declare variables $\boldsymbol{V}$ and functions $\boldsymbol{F}$
- and the body statement is well-formed with respect to those global variables $\boldsymbol{V}$ and functions $\boldsymbol{F}$
  - all variables and functions are declared
  - functions are called with the correct number of arguments

Static Semantics

$$\frac{\vdash \textit{gdecs}\ \textit{decs}\ V,\ F \qquad V,F \vdash \textit{stmt}\ \textit{ok}}{\vdash \textit{gdecs}\ \textit{stmt}\ \textit{ok}}$$

gdecs

```
int result = 0;
int div (int x, int y) {
  int res = 0;
  while (x > y) {
    x = x - y;
    res = res + 1;
  }
  return res;
}
```

stmt

```
result = div (16, 5);
```

$$\vdash \textit{gdecs}\ \textit{decs}\ \underbrace{\{\texttt{result}\}}_{V}\underbrace{\{\texttt{div:2}\}}_{F}$$

$$\{\texttt{result}\}\{\texttt{div:2}\} \vdash \texttt{result = div (16, 5)}\ \textit{ok}$$

Static Semantics

- Well-formedness of statements: a statement is well-formed if all its constituents are well-formed:

$$\frac{V,F \vdash expr\ ok \qquad V,F \vdash expr\ ok}{V,F \vdash \texttt{while}\ (expr)\ stmt\ ok}$$

$$\frac{V,F \vdash expr\ ok \quad V,F \vdash stmt_1\ ok \quad V,F \vdash stmt_1\ ok}{V,F \vdash \texttt{if}\ (expr)\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ ok}$$

- Well-formedness of a block:

$$\frac{V,F \vdash ldecs\ decs\ V' \qquad V' \cup V,F \vdash stmts\ ok}{V,F \vdash \{\ ldecs\ stmts\ \}\ ok}$$

Utrecht University

Static Semantics

$$\underbrace{}_{ldecs}\quad\underbrace{}_{stmts}$$

```
{
  int res = 0;
  while (x > y) {
    x = x - y;
    res = res + 1;
  }
  return res;
}
```

$$\frac{V,F \vdash ldecs\ \ decs\ \ V' \qquad V' \cup V,F \vdash\ stmts\ \ ok}{V,F \vdash \{\ ldecs\ \ stmts\ \}\ ok}$$

$$\frac{\{x,y\},\{div:2\} \vdash ldecs\ ok \quad \{x,y,res\},\{div:2\} \vdash stmts\ ok}{\{x,y\},\{div:2\} \vdash \{\ ldecs\ \ stmts\ \}\ ok}$$

Static Semantics

- Well-formedness of expressions: an expression is well-formed if
  - all of its variables and functions are declared and
  - functions are called with the correct number of arguments

$$\frac{x \in V}{V, F \vdash x \ ok}$$

$$\frac{V, F \vdash expr \ ok \qquad x \in V}{V, F \vdash x = expr \ ok}$$

$$\frac{V, F \vdash expr_i \ ok \qquad f : n \in F}{V, F \vdash f(expr_1, \ldots expr_n) \ ok}$$

Static Semantics

- Well-formedness of individual declarations:
  - variable declarations (variable names have to be unique in the scope)

$$\frac{x \notin V}{V,\ F \vdash\ \texttt{int}\ x\ \texttt{=}\ v\texttt{;}\ \mathit{decs}\{x\},\ \varnothing}$$

  - function declarations (function names, formal parameter names unique in the scope)

$$\frac{x_i \notin V \quad f \notin F \quad V \cup \{x_1,\ldots\ x_n\},\ F \cup \{f{:}n\} \vdash_{\mathit{stmt}} \mathit{ok}}{V,\ F \vdash\ \texttt{int}\ f\ \texttt{(int}\ x_1\texttt{,}\ldots\texttt{,int}\ x_n\texttt{)}\ \mathit{stmt}\ \mathit{decs}\ \ \varnothing,\{f{:}n\}}$$

Dynamic Semantics

- we're describing TinyC's semantics using a big step semantics relation $\Downarrow$
- we have to think about how to model the execution of a program in this framework
- how would we trace the execution on paper?

```
int result = 0;

int offset = 10;

int f (int z) {
  return (z + offset);
}

int div (int x, int y) {
  int res = 0;
  while (x > y) {
    x = x - y;
    res = res + 1;
  }
  return res;
}

result = div (16, 5);
```

Big Step Dynamic Semantics

- What information do we have to keep track of?
  - the current statement
  - the values of all variables which are in scope
    - in MinHs, we substituted the value of a variable. We can't do that in TinyC!

```
int result = 1;
int z      = 0;
int f(int x) {
 if (x > 0) {
    x = x - 1;
    result =
      2 * result;
    f(x);
    return x;
  } else {}
  return x;
}
z = f (3);
```

f(0)	x	0	
f(1)	x	~~1~~	0
f(2)	x	~~2~~	1
f(3)	x	~~3~~	2
	z	~~0~~	2
	result	~~1~~	~~2~~ ~~4~~ 8
	int f(int…		

Big Step Dynamic Semantics

- Machine state needs to contain the current memory state (including code for functions), and current expression/statement:

$$\left(\begin{array}{l}\texttt{int result = 0;}\\ \texttt{int offset = 10;}\\ \texttt{int div(int …)\{\}}\end{array} , \texttt{result = offset + 1} \right) \Downarrow \left(\begin{array}{l}\texttt{int result = 11;}\\ \texttt{int offset = 10;}\\ \texttt{int div(int …)\{\}}\end{array} , \texttt{11} \right)$$

- Evaluation relations

  - program execution $(g\ \ s) \Downarrow (g', \ rv)$

  - statement execution: $(g, \ s) \Downarrow (g', \ rv)$

  - expression execution: $(g, \ e) \Downarrow (g', \ v)$

where v is a integer value, rv either a integer value or $\texttt{return}(v)$

Big Step Dynamic Semantics

- The environment is an ordered sequence (stack in our example) associating
  - variables with integer values
  - function names with parameter list and body
- Operations on the environment:
  - extension:
    - add a new declaration `(int x = 4)` to the environment g:
      - g `. (int x = 4)`

we're overloading the '.' symbol here!
has nothing to do with '.' to denote
a bound variable in HO abstract syntax

  - lookup of variable values: $g@x =$ `5`
    - x is currently bound to value `5`
    - if x occurs more than once, choose right most binding (upper most in the diagram). Important that names are unique here!
  - lookup of function declaration: $g@f =$ `int` f `(int` $x_1, \ldots$ `int` x_n`)` s
  - update of variable values: $g@x \leftarrow$ `5`
    - change value of (the right most) x to `5`

Utrecht University

Big Step Dynamic Semantics

- program execution $(g \;\; s) \Downarrow (g', \;\; rv)$

$$\frac{(g \;\;,\;\; s) \Downarrow (g', \;\; rv)}{(g \;\; s) \Downarrow (g', \;\; rv)}$$

Dynamic Semantics

- if-statements:

$$\frac{(g,\ e)\Downarrow(g',0) \qquad (g',\ s_2)\Downarrow(g'',rv)}{(g,\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\Downarrow(g'',rv)}$$

$$\frac{(g,\ e)\Downarrow(g',v) \quad v\neq 0 \qquad (g',\ s_1)\Downarrow(g'',rv)}{(g,\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\Downarrow(g'',rv)}$$

- while-loops:

$$\frac{(g,\ e)\Downarrow(g',0)}{(g,\ \texttt{while}\ (e)\ s)\Downarrow(g',0)}$$

$$\frac{(g,\ e)\Downarrow(g',v) \qquad (g',\ s;\ \texttt{while}\ (e)\ s)\Downarrow(g'',rv)}{(g,\ \texttt{while}\ (e)\ s)\Downarrow(g'',rv)}$$

- alternatively, in terms of if-statements:

$$\frac{(g,\ \texttt{if}\ e\ \texttt{then}\ \{s;\texttt{while}\ (e)\ s\}\ \texttt{else}\ 0)\Downarrow(g'',rv)}{(g,\ \texttt{while}\ (e)\ s)\Downarrow(g'',rv)}$$

Dynamic Semantics

- Return statements:

$$\frac{(g,\ e) \Downarrow (g',v)}{(g,\ \texttt{return}(e)) \Downarrow (g',\texttt{return}(v))}$$

- Blocks:

$$\frac{(g.l,\ ss) \Downarrow (g'.l',\ rv)}{(g,\ \{l\ ss\}) \Downarrow\ (g',\ rv)}$$

add local variables with initial value temporarily into environment, only g is threaded through – the local environment l/l' is discarded after the statements ss are executed

Utrecht University

Dynamic Semantics

- Sequence of statements:

$$\frac{}{(g,\ \circ)\Downarrow(g,0)}$$

$$\frac{(g,\ s)\Downarrow(g',v;) \qquad (g',\ ss)\Downarrow(g'',v')}{(g,\ s\ ss)\Downarrow(g'',v')}$$

$$\frac{(g,\ s)\Downarrow(g',\ \texttt{return}(v))}{(g,\ s\ ss)\Downarrow(g',\texttt{return}(v))}$$

- Variables:

$$\frac{g@x = v}{(g,\ x)\Downarrow(g,\ v)}$$

$$\frac{(g,\ e)\Downarrow(g',v)}{(g,\ x{=}e)\ \Downarrow\ (g'@x\leftarrow v,\ v)}$$

Utrecht University

Dynamic Semantics

- Function calls:

$$\frac{g@f = \texttt{int}\ f\,(\texttt{int}\ x_1,\ldots\ \texttt{int}\ x_n)\,s \qquad (g,(e_1,\ldots,e_n)) \Downarrow (g',(v_1,\ldots,v_n)) \qquad (g',\{\texttt{int}\ x_1 = v_1;\ldots;s\}) \Downarrow (g'',\texttt{return}(v))}{(g, f\,(e_1,\ldots,e_n)) \Downarrow (g'', v)}$$

$$\frac{g@f = \texttt{int}\ f\,(\texttt{int}\ x_1,\ldots\ \texttt{int}\ x_n)\,s \qquad (g,(e_1,\ldots,e_n)) \Downarrow (g',(v_1,\ldots,v_n)) \qquad (g',\{\texttt{int}\ x_1 = v_1;\ldots;s\}) \Downarrow (g'',v)}{(g, f\,(e_1,\ldots,e_n)) \Downarrow (g'', v)}$$

$(g, (e_1,\ldots,e_n)) \Downarrow (g', (v_1,\ldots,v_n))$ what about the evaluation order?

```
int result = 1;
int z      = 0;
int f(int x) {
 if (x > 0) {
    x = x - 1;
    result =
      2 * result;
    f(x);
    return x;
  } else {}
  return x;
 }
z = f (2);
```

$$g@f = \texttt{int}\ f\,(\texttt{int}\ x_1, \ldots\ \texttt{int}\ x_n)\,s$$

$$\frac{(g, (e_1,\ldots,e_n)) \Downarrow (g', (v_1,\ldots,v_n)) \qquad (g', \{\texttt{int}\ x_1 = v_1; \ldots; s\}) \Downarrow (g'', \texttt{return}(v))}{(g, f\,(e_1,\ldots,e_n)) \Downarrow (g'', v)}$$

```
int x = 2;
int result = 1;
int z      = 0;
f (int x) {…}
```

`x = x-1; result = …; f(x)` $\Downarrow$

```
int x = 2;
int result = 1;
int z      = 0;
f (int x) {…}
```

`if (x > 0) {x = x-1; result = …; f(x)` $\Downarrow$

```
int result = 1;
int z      = 0;
f (int x) {…}
```

`{int x = 2; if (x > 0)…` $\Downarrow$

```
int result = 1;
int z      = 0;
f (int x) {…}
```

`f(2)` $\Downarrow$

$$\frac{(g.l, ss) \Downarrow (g'.l', rv)}{(g, \{l\ ss\}) \Downarrow (g', rv)}$$

$$\frac{(g, e) \Downarrow (g', v)}{(g, x = e;) \Downarrow (g'@x \leftarrow v, v)}$$

```
int result = 1;
int z      = 0;
f (int x) {…}
```

`z = f(2)` $\Downarrow$

Utrecht University

```
int result = 1;
int z      = 0;
int f(int x) {
 if (x > 0) {
    x = x - 1;
    result =
      2 * result;
    f(x);
    return x;
  } else {}
  return x;
 }
z = f (2);
```

```
int x = 2;
```

```
int result = 1;
int z      = 0;
f (int x) {…}
```

```
x = x-1; result = …; f(x) ⇓
```

```
int x = 2;
```

```
int result = 1;
int z      = 0;
f (int x) {…}
```

```
if (x > 0) {x = x-1; result = …; f(x) ⇓
```

```
int result = 1;
int z      = 0;
f (int x) {…}
```

```
{int x = 2; if (x > 0)… ⇓
```

```
int result = 1;
int z      = 0;
f (int x) {…}
```

```
f(2) ⇓
```

```
int result = 1;
int z      = 0;
f (int x) {…}
```

```
z = f(2) ⇓
```

```
int result = 1;
int z       = 0;
int f(int x) {
 if (x > 0) {
    x = x - 1;
    result =
      2 * result;
    f(x);
    return x;
  } else {}
  return x;
 }
z = f (2);
```

$$\frac{(g,\ e) \Downarrow (g',v)}{(g,\ x{=}e;) \Downarrow (g'@x \leftarrow v,\ v)}$$

```
int x = 0;
int x = 1;
int result = 4;
int z        = 0;
f (int x) {…}
```

`f(x)` ⇓

```
int x = 0;
int x = 1;
int result = 4;
int z        = 0;
f (int x) {…}
```

`0`

```
int x = 1;
int x = 1;
int result = 2;
int z        = 0;
f (int x) {…}
```

`x = x-1;…; f(x); return x` ⇓

```
int x = 0;
int x = 1;
int result = 4;
int z        = 0;
f (int x) {…}
```

`0`

```
int x = 1;
int result = 2;
int z        = 0;
f (int x) {…}
```

`f(x)` ⇓

```
int x = 1;
int result = 4;
int z        = 0;
f (int x) {…}
```

`1`

```
int x = 2;
int result = 1;
int z        = 0;
f (int x) {…}
```

`x = x-1;…; f(x); return x` ⇓

```
int x = 1;
int result = 4;
int z        = 0;
f (int x) {…}
```

`1`

```
int result = 1;
int z       = 0;
int f(int x) {
 if (x > 0) {
    x = x - 1;
    result =
      2 * result;
    f(x);
    return x;
  } else {}
  return x;
 }
z = f (2);
```

```
int x = 2;
int result = 1;
int z      = 0;
f (int x) {…}
```
`x = x-1; result = …; f(x)` ⇓
```
int x = 1;
int result = 4;
int z      = 0;
f (int x) {…}
```
`1`

```
int x = 2;
int result = 1;
int z      = 0;
f (int x) {…}
```
`if (x > 0) {…f(x)}` ⇓
```
int x = 1;
int result = 4;
int z      = 0;
f (int x) {…}
```
`1`

```
int result = 1;
int z      = 0;
f (int x) {…}
```
`{int x = 2; if (x > 0)…` ⇓
```
int result = 4;
int z      = 0;
f (int x) {…}
```
`1`

```
int result = 1;
int z      = 0;
f (int x) {…}
```
`f(2)` ⇓
```
int result = 4;
int z      = 0;
f (int x) {…}
```
`1`

```
int result = 1;
int z      = 0;
f (int x) {…}
```
`z = f(2)` ⇓
```
int result = 4;
int z      = 1;
f (int x) {…}
```
`1`

Utrecht University

Utrecht University

Concepts of Programming Language Design

Reference Types

Gabriele Keller
Tom Smeding

Reference types

- Variables in TinyC represent values stored in a fixed memory location
  - assigning a new value to a variable updated the value in that location
- Reference types refer to a location a value is stored
- Reference types are usually implemented as pointers, that is as address into the memory of a process (often with some associated meta data, such as the size of the data pointed to)
- Most high-level languages support or use reference types in one way or another
  - explicitly, exposing the implementation as pointer: C
  - explicitly, in an abstract way: only expose the interface (creation, read and write a value)
  - implicitly, using them behind the scenes to implement data structures

Utrecht University

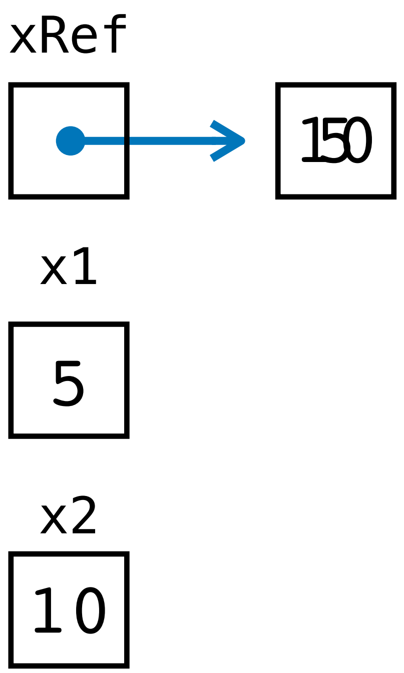
Reference types

- Haskell has not explicit built-in reference types, but Data.IORef provides it as abstract data type:

```haskell
newIORef   :: a                -> IO (IORef a)
writeIORef :: a -> IORef a -> IO ()
readIORef  :: IORef            -> IO a
```

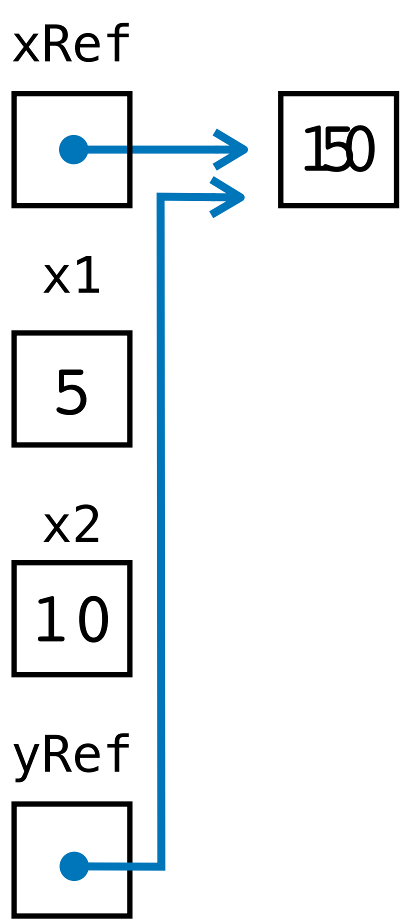
- these are functions which have an effect on the world (or depend on the current state of the world)

```haskell
main = do
  xRef <- newIORef 5
  x1 <- readIORef xRef
  writeIORef xRef 10
  x2 <- readIORef xRef
  putStrLn ("x1: " ++ (show x1) ++ " x2: " ++ (show x2))
```



Reference types

```haskell
main = do
  xRef <- newIORef 5
  x1 <- readIORef xRef
  let yRef = xRef
  writeIORef yRef 10
  x2 <- readIORef xRef
  putStrLn ("x1: " ++ (show x1) ++ " x2: " ++ (show x2))
```

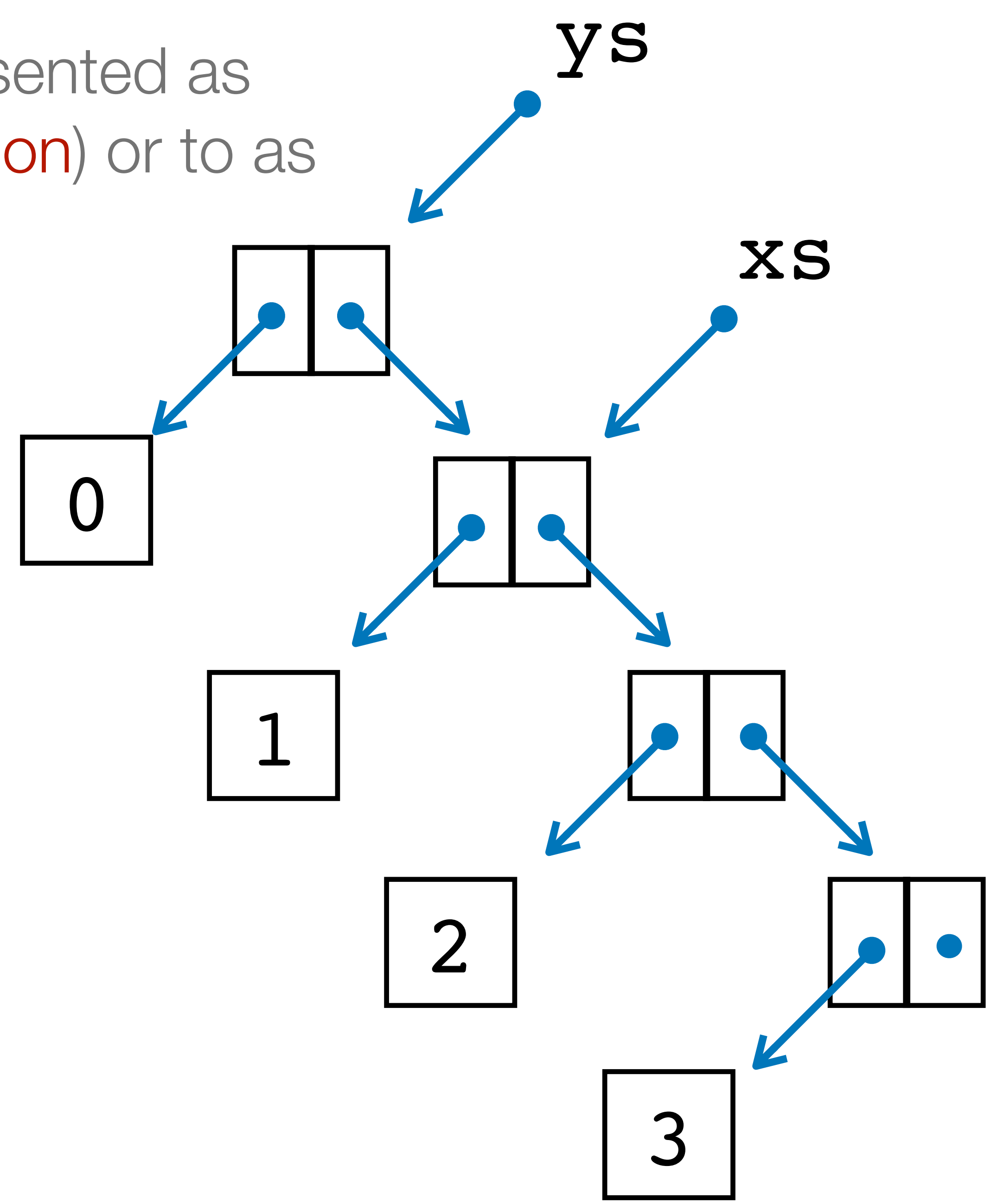


x1: 5 x2: 10

Reference types

- Haskell also uses references behind the scenes
  - even basic values (Int etc) are internally represented as references to these values (boxed representation) or to as to yet unevaluated computations
  - enables sharing

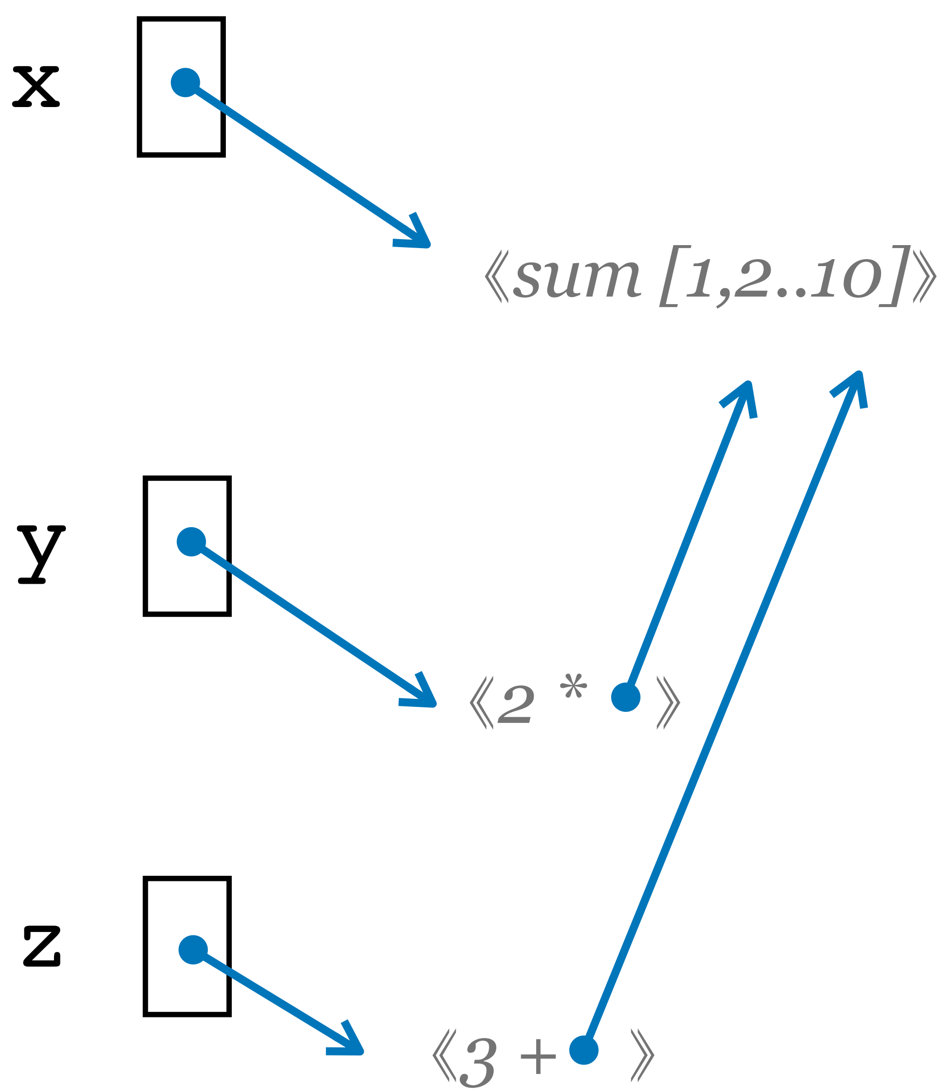
```haskell
let
  xs = [1,2,3]
  ys = 0 : xs
```



Reference types

- The boxed representation is an effective representation for sharing (lazy evaluation!)

```
let
  x = sum [1,2..10]
  y = 2 * x
  z = 3 + x
```

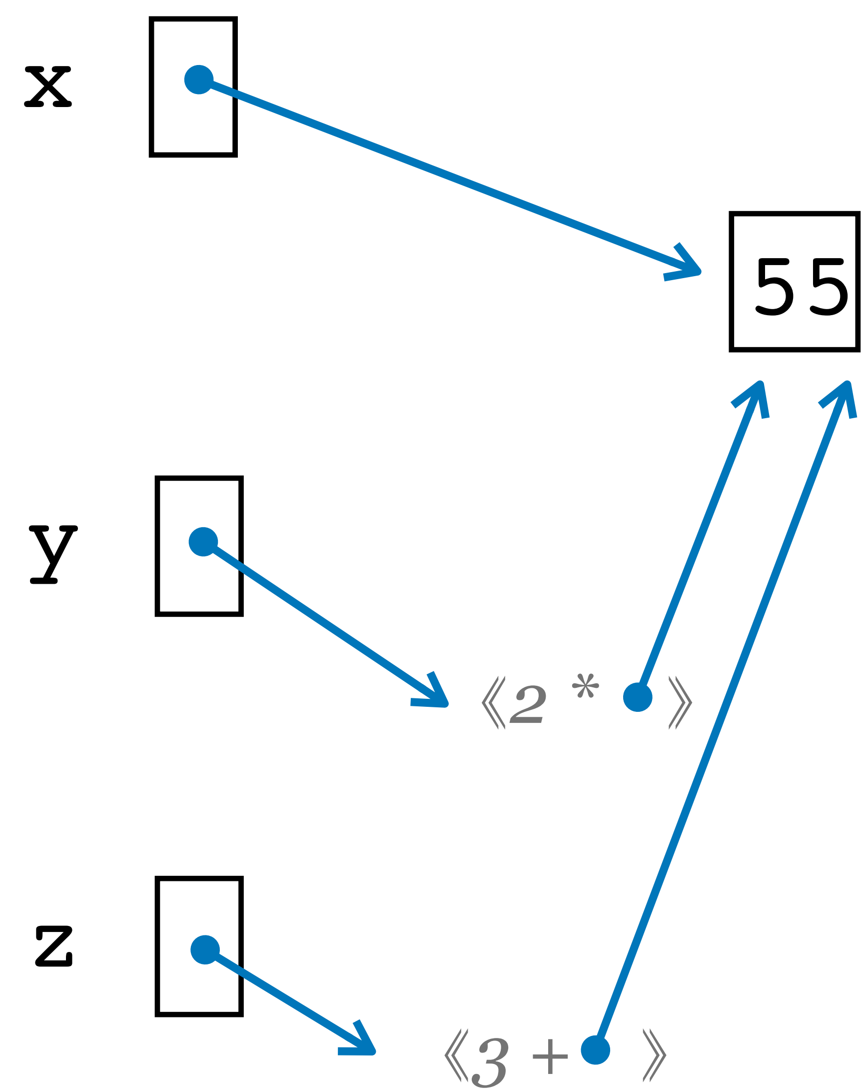


- This means evaluation has a side effect (this can be problematic for parallel execution)

Reference types

- The boxed representation is an effective representation for sharing (lazy evaluation!)

```
let
  x = sum [1,2..10]
  y = 2 * x
  z = 3 + x
```

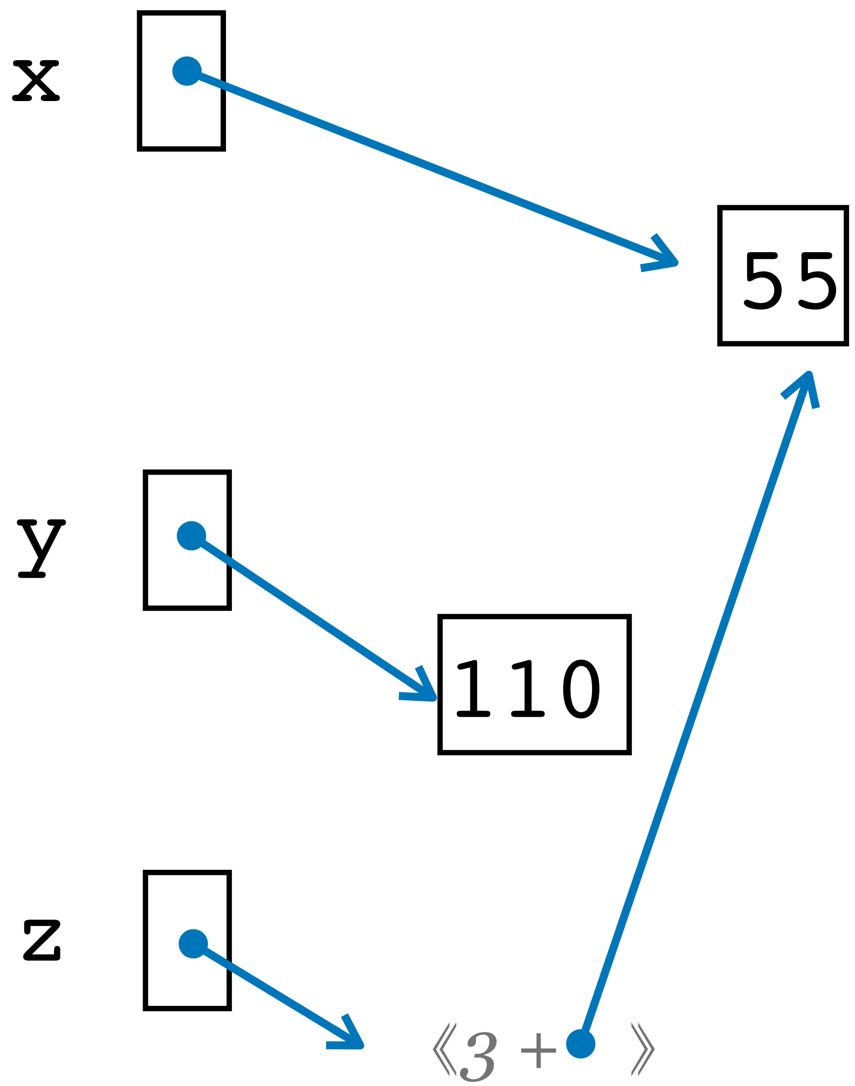


- This means evaluation has a side effect (this can be problematic for parallel execution)

Reference types

- The boxed representation is an effective representation for sharing (lazy evaluation!)

```
let
  x = sum [1,2..10]
  y = 2 * x
  z = 3 + x
```



- This means evaluation has a side effect (this can be problematic for parallel execution)

Reference types

- In functional languages, it doesn't matter for the semantics of a program whether a value has a boxed or unboxed representation
  - it does affect performance, as dereferencing is expensive
  - in Haskell, it's possible to explicitly use unboxed types (denoted by # - `Int#`…)

References in stateful languages

- In languages with side effect, it is important to know whether we deal with reference or value types to understand the behaviour of
  - function calls
  - assignments
- Unfortunately, this is not uniform, even across closely related languages

```
public class MyClass
{
  public int value;
}

public class Program
{
  public static void Main()
  {
    MyClass ob1 = new MyClass();
    ob1.value   = 20;
    MyClass ob2 = ob1;
    ob1.value   = 10;
    Console.WriteLine("ob2.value = {0}", ob2.value);
  }
```

```
class MyClass
{
  public: int value;
};

int main() {
    MyClass ob1;
    ob1.value   = 20;
    MyClass ob2 = ob1;
    ob1.value   = 10;
    std::cout << "obj2.value = "<< ob2.value;
    return 0;
}
```

Check out differences in value & reference type classification when switching to a new language!

Language	Value type	Reference type
C++[3]	booleans, characters, integer numbers, floating-point numbers, classes (including strings, lists, maps, sets, stacks, queues), enumerations	references, pointers
Java[4]	booleans, characters, integer numbers, floating-point numbers	arrays, classes (including immutable strings, lists, dictionaries, sets, stacks, queues, enumerations), interfaces, null pointer
C#[5]	structures (including booleans, characters, integer numbers, floating-point numbers, point in time i.e. DateTime, optionals i.e. Nullable<T>), enumerations	classes (including immutable strings, arrays, tuples, lists, dictionaries, sets, stacks, queues), interfaces, pointers
Swift[6][7]	structures (including booleans, characters, integer numbers, floating-point numbers, fixed-point numbers, mutable strings, tuples, mutable arrays, mutable dictionaries, mutable sets), enumerations (including optionals), and user-defined structures and enumerations composing other value types.	functions, closures, classes
Python[8]		classes (including immutable booleans, immutable integer numbers, immutable floating-point numbers, immutable complex numbers, immutable strings, byte strings, immutable byte strings, immutable tuples, immutable ranges, immutable memory views, lists, dictionaries, sets, immutable sets, null pointer)
JavaScript[9]	immutable booleans, immutable floating-point numbers, immutable integer numbers (bigint), immutable strings, immutable symbols, undefined, null	objects (including functions, arrays, typed arrays, sets, maps, weak sets and weak maps)
OCaml[10][11]	immutable characters, immutable integer numbers, immutable floating-point numbers, immutable tuples, immutable enumerations (including immutable units, immutable booleans, immutable lists, immutable optionals), immutable exceptions, immutable formatting strings	arrays, immutable strings, byte strings, dictionaries (including pointers)

https://en.wikipedia.org/wiki/Value_type_and_reference_type

Call-by-value vs call-by reference

- Also called pass-by-reference/pass-by-value
- What is the calling convention for procedures/functions/methods?
- Call by value
  - like in TinyC (and C): the value of the argument expression gets bound to the formal parameter.
  - function calls don't affect the values of the variables in the caller
- Java, C#, C++ are all call by value, but since classes are reference types in C# & Java, the behaviour is different (the reference gets copied, in C++, the object gets copied)

Utrecht University

Call-by-value vs call-by reference

```
void swap1 (int x, int y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}


void swap2 (int * x, int * y) {
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

int a = 5;
int b = 7;

swap1 (a, b);
swap2 (&a, &b);
```

Call-by-value vs call-by reference

- Fortran is always call by reference - even on constant values!

- Java, C#, C++ are all call by value, but since classes are reference types in C# & Java, the behaviour is different (the reference gets copied, in C++, the object gets copied)

Adding references

$$
\begin{array}{lcl}
\mathit{prgm} & ::= & \mathit{gdecs}\ \mathit{rdecs}\ \mathit{stmt} \\
\mathit{gdecs} & ::= & \epsilon \mid \mathit{gdec}\ \mathit{gdecs} \\
\mathit{gdec} & ::= & \mathit{fdec} \mid \mathit{vdec} \\
\mathit{vdecs} & ::= & \epsilon \mid \mathit{vdec}\ \mathit{vdecs} \\
\mathit{type} & ::= & \texttt{int}\ \mathit{Ident} \mid \texttt{int} * \mathit{Ident} \\
\mathit{vdec} & ::= & \mathit{type}\ \texttt{=}\ v\ \texttt{;} \\
\mathit{rdecs} & ::= & \epsilon \mid \mathit{rdec}\ \mathit{rdecs} \\
\mathit{rdec} & ::= & \texttt{int} *\ \mathit{Ident}\texttt{=}\ \texttt{alloc(}v\texttt{);} \\
\mathit{fdec} & ::= & \mathit{type}\ \mathit{Ident}_2\ \texttt{(}\mathit{arguments}\texttt{)}\ \mathit{stmt} \\
\mathit{stmt} & ::= & \mathit{expr}\ \texttt{;} \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\texttt{;} \mid \texttt{return}\ \mathit{expr}\ \texttt{;} \mid \\
 & & \{\ \mathit{vdecs}\ \mathit{rdecs}\ \mathit{stmts}\ \} \mid \texttt{while (}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt} \\
\mathit{stmts} & ::= & \epsilon \mid \mathit{stmt}\ \mathit{stmts} \\
\mathit{expr} & ::= & \mathit{Num} \mid \mathit{Ident} \mid *\ \mathit{Ident} \mid \mathit{expr}_1\ \texttt{+}\ \mathit{expr}_2 \mid \mathit{expr}_1\ \texttt{-}\ \mathit{expr}_2 \mid \\
 & & \mathit{Ident}\ \texttt{=}\ \mathit{expr} \mid *\mathit{Ident}\ \texttt{=}\ \mathit{expr} \mid \mathit{Ident}\ \texttt{(}\mathit{exprs}\texttt{)} \\
\mathit{arguments} & ::= & \epsilon \mid \mathit{type}\ \mathit{Ident}_2\ \texttt{,}\ \mathit{arguments}
\end{array}
$$

Adding Pointers

- What is a pointer?