Creating Language Processors

Paul Klint

Agenda

- Understanding how to write tools for the EXP language.
- The Pico language:
  - Concrete Syntax
  - Abstract Syntax
  - Type checking
  - Assembly Language
  - Compilation to Assembly Language
  - Embedding in Eclipse

Concrete *vs* Abstract Syntax

Concrete syntax:

```
lexical LAYOUT = [\t-\n\r\ ];

lexical IntegerLiteral = [0-9]+;

start syntax Exp
       = con: IntegerLiteral
       | bracket "(" Exp ")"
       > left mul: Exp "*" Exp
       > left add: Exp "+" Exp
       ;
```

Abstract syntax:

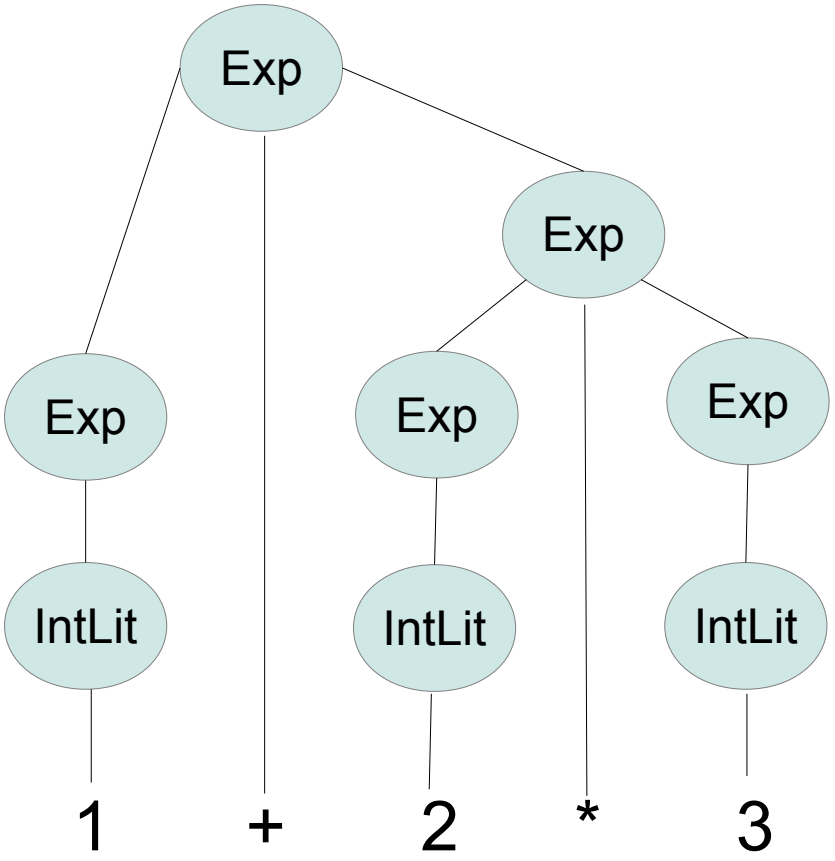
```
data AExp = con(int n)
          | mul(AExp e1, AExp e2)
          | add(AExp e1, AExp e2)
          ;
```

- Results in a parse tree
- Contains all textual information
- Much “redundant” information

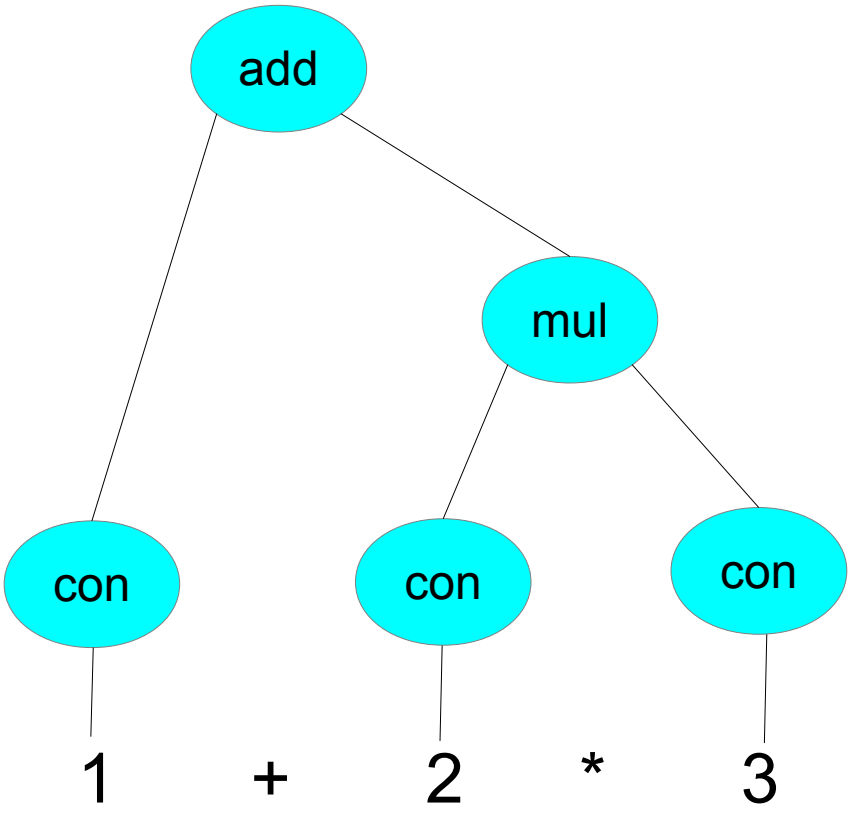
- More dense representation
- Contains “essential” information
- More convenient for tools

Parse Tree *vs* Abstract Syntax Tree

Parse Tree (PT):



Abstract Syntax Tree (AST):



Recall Matching of Abstract Patterns

```
public int invertRed(CTree t) {
  return visit(t){
              case red(CTree t1, CTree t2) =>
                    red(t2, t1)
  };
}
```

Concrete Patterns

- A text fragment of a defined language between ` and ` (backquotes).
- Prefixed with (NonTerminal), e.g. (Exp)
- May contain holes, e.g. <Exp e1>
- Matches the parse tree for the text fragment (and binds any patterns variables)
- Example: (Exp) `<Exp e1> * <Exp e2>`
- Example: (Exp) `IntegerLiteral l>`

From PT to AST: Manual

```
public Exp parse(str txt) = parse(#Exp, txt);

public AExp load(str txt) = load(parse(txt));
```

```
public AExp load((Exp)`<IntegerLiteral l>`)
       = con(toInt("<l>"));
public AExp load((Exp)`<Exp e1> * <Exp e2>`)
       = mul(load(e1), load(e2));
public AExp load((Exp)`<Exp e1> + <Exp e2>`)
       = add(load(e1), load(e2));
public AExp load((Exp)`( <Exp e> )`)
       = load(e);
```

```
rascal>load("1+2")
AExp: add(
  con(1),
  con(2))
```

From PT to AST: Automatic

```
public AExp load2(str txt) = implode(#AExp, parse(txt));
```

```
rascal>load2("1 + 2")
AExp: add(
  con(1)[
    @location=|file://-|(0,1,<1,0>,<1,1>)
  ],
  con(2)[
    @location=|file://-|(4,1,<1,4>,<1,5>)
  ])[
  @location=|file://-|(0,5,<1,0>,<1,5>)
]
```

implode:

- Automatically maps the parse tree to the given ADT (AExp)
- Preserves location information (location annotation) that can be used in error messages

An EXP Evaluator

- Goal: a function `public int eval(str txt)` that
  - Takes a string (hopefully a syntactically correct EXP sentence)
  - Evaluates it as an expression and returns the result
  - `eval("2 + 3") => 5`
- How to implement `eval`?

An EXP Evaluator

```
public int eval(con(int n)) = n;
public int eval(mul(AExp e1, AExp e2)) = eval(e1) * eval(e2);
public int eval(add(AExp e1, AExp e2)) = eval(e1) + eval(e2);
```

```
public int eval(str txt) = eval(load(txt));
```

```
rascal>eval("1+2*3")
int: 7
```

EXP Statistics

Given:

- `alias` Stats = `tuple`[`int` addcnt, `int` mulcnt];

- Goal: a function `public Stats calcStats(str txt)` that
  - Takes a string
  - Counts all + and * operators
  - `calcStats("2 + 3") => <1, 0>`
- How to implement `calcStats`?

EXP Statistics

```
alias Stats = tuple[int addcnt, int mulcnt];

public Stats calcStats(con(int n), Stats s) = s;

public Stats calcStats(mul(AExp e1, AExp e2), Stats s) {
       s1 = calcStats(e2, calcStats(e1, s));
       return <s1.addcnt, s1.mulcnt +1>;
}

public Stats calcStats(add(AExp e1, AExp e2), Stats s) {
       s1 = calcStats(e2, calcStats(e1, s));
       return <s1.addcnt + 1, s1.mulcnt>;
}

public Stats calcStats(str txt) = calcStats(load2(txt), <0,0>);
```

```
rascal>calcStats("1+2+3*4+5")
tuple[int addcnt,int mulcnt]: <3,1>
```

An EXP Unparser

- Goal: a function `public str unparse(AExp e)` that
  - transforms an AST into a string.
  - Satisfies the equality load(parse(unparse(t))) == t.
- This function is mostly useful to show the textual result after a program transformation

An EXP Unparser

```
public str unparse(con(int n)) = "<n>";

public str unparse(mul(AExp e1, AExp e2))
       = "<unparse(e1)> * <unparse(e2)>";

public str unparse(add(AExp e1, AExp e2))
       = "<unparse(e1)> + <unparse(e2)>";

public str unparse(str txt) = unparse(load2(txt));
```

```
rascal>unparse("1 + 2")
str: "1 + 2"
```

Is load(parse(unparse(t))) == t satisfied?

No!

```
rascal>unparse("1 + 2 * (3 + 4)")
str: "1 + 2 * 3 + 4"
```

Improved EXP Unparser

```
public str unparse2(con(int n)) = "<n>";

public str unparse2(mul(AExp e1, AExp e2))
       = "<unparseBracket(e1)> * <unparseBracket(e2)>";

public str unparse2(add(AExp e1, AExp e2))
       = "<unparse2(e1)> + <unparse2(e2)>";

public str unparseBracket(add(AExp e1, AExp e2))
       = "( <unparse2(e1)> + <unparse2(e2)> )";
public default str unparseBracket(AExp e) = unparse2(e);

public str unparse2(str txt) = unparse2(load2(txt));
```

```
rascal>unparse2("1 + 2 * (3 + 4)")
str: "1 + 2 * ( 3 + 4 )"
```

An unparser should be aware of syntactic priorities

EXP RPN Translator

- RPN = Reverse Polish Notation
- Every operator follows its operands
- Invented by Polish logician Jan Łukasiewicz
- Used in HP calculators
- Stack-based language Forth

RPN Examples

Infix:

- 3
- 3 + 4
- 3 + 4 * 5
- 3 + 4 + 5 * (6 + 7)

RPN:

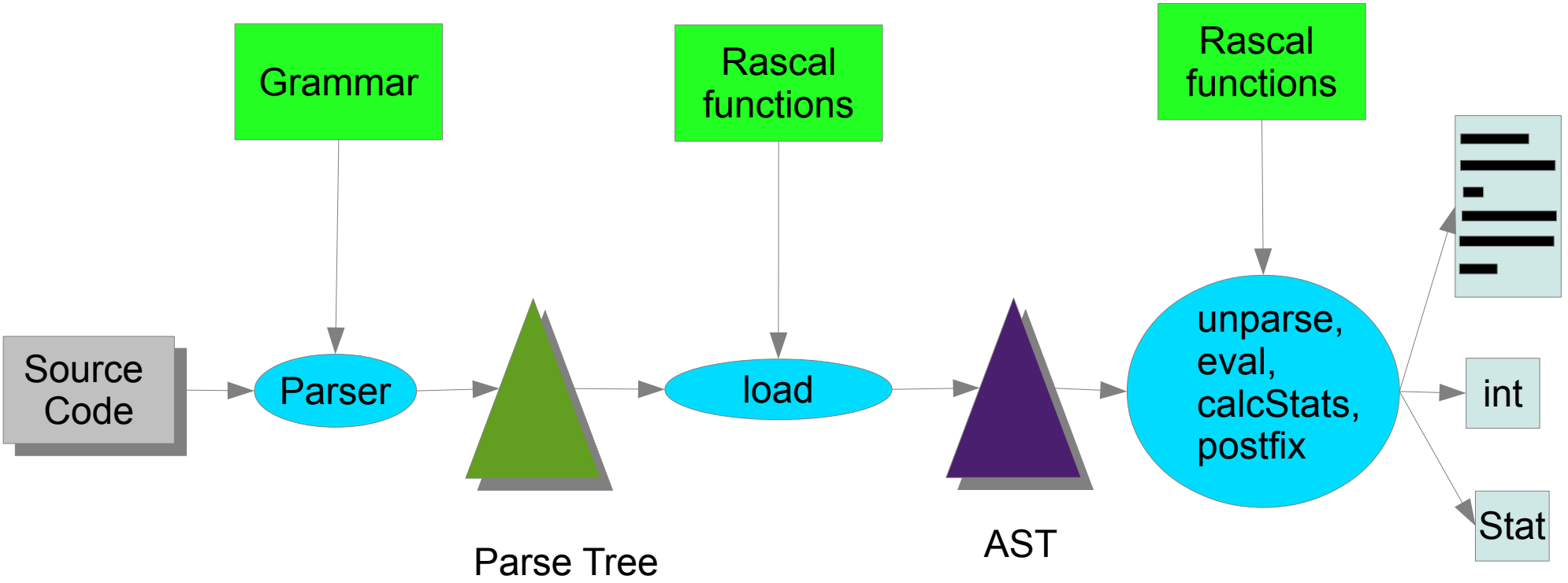
- 3
- 3 4 +
- 3 4 5 * +
- 3 4 + 5 6 7 + * +

EXP RPN Translator

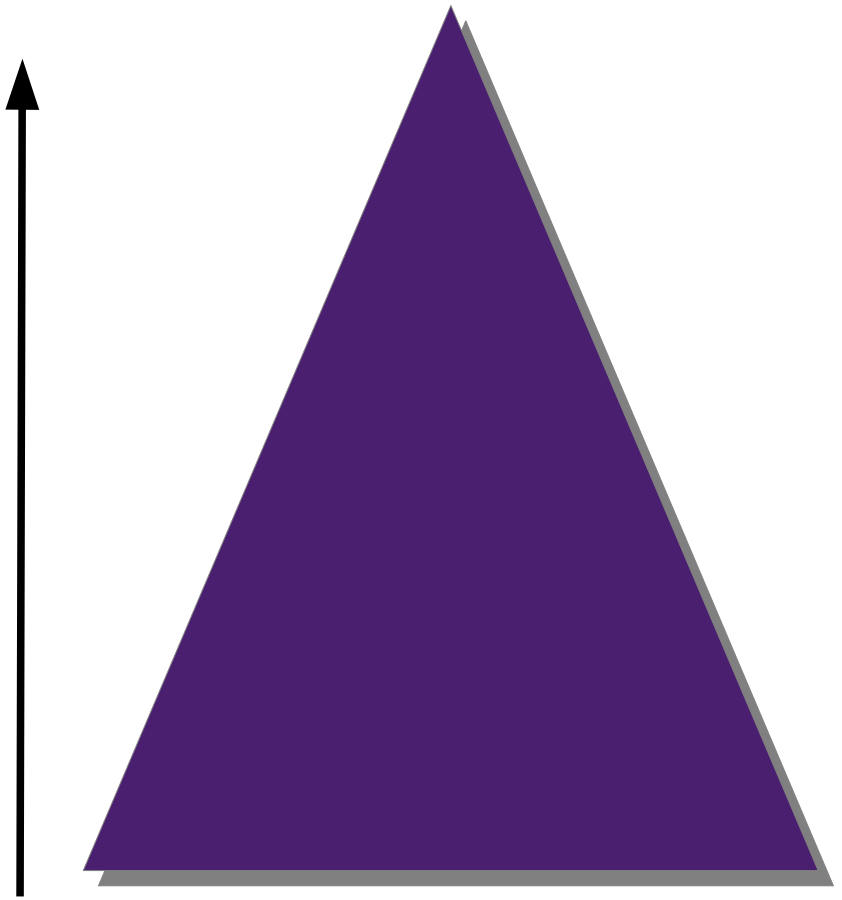
```
public str postfix(con(int n)) = "<n>";

public str postfix(mul(AExp e1, AExp e2))
       = "<postfix(e1)> <postfix(e2)> *";

public str postfix(add(AExp e1, AExp e2))
       = "<postfix(e1)> <postfix(e2)> +";

public str postfix(str txt) = postfix(load2(txt));
```

```
rascal>postfix("3 + 4  + 5 * (6 + 7)")
str: "3 4 + 5 6 7 + * +"
```

Commonalities, 1



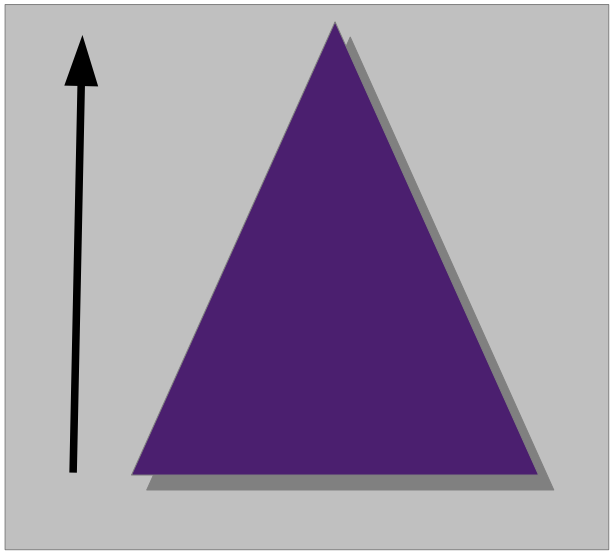
Commonalities, 2

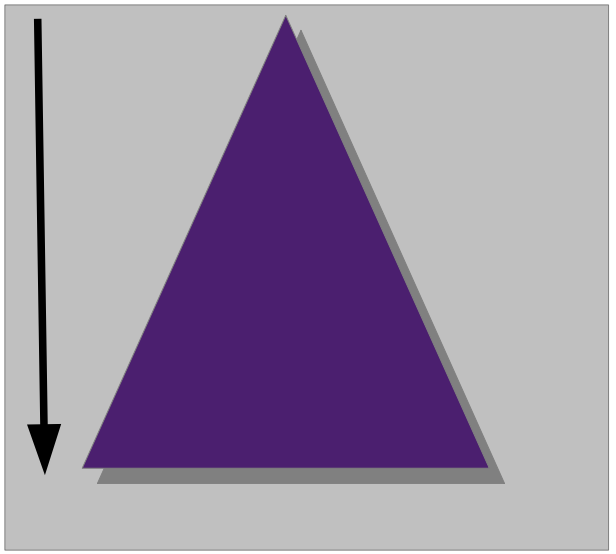


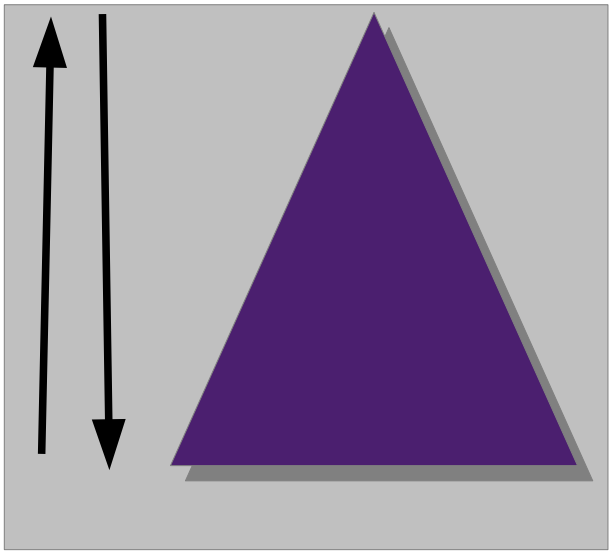
Bottom-up information flow ("*synthesized*")

- Eval results
- Unparsed strings
- Stat values
- ...

Commonalities, 3







Pure synthesized:
- Context-free
- Subtrees treated independently

Pure inherited:
- Rare

Inherited and synthesized
- Context-dependent
- Subtrees can influence each other
- Essential for
  - Typechecking declarations
  - Evaluating variables

The Toy Language Pico

- Has a single purpose: being so simple that its specification fits on a few pages
- We will define various operations on Pico programs:
  - Parse
  - Typecheck
  - Compile to Assembler
- We will integrate the Pico tools with Eclipse

The Toy Language Pico

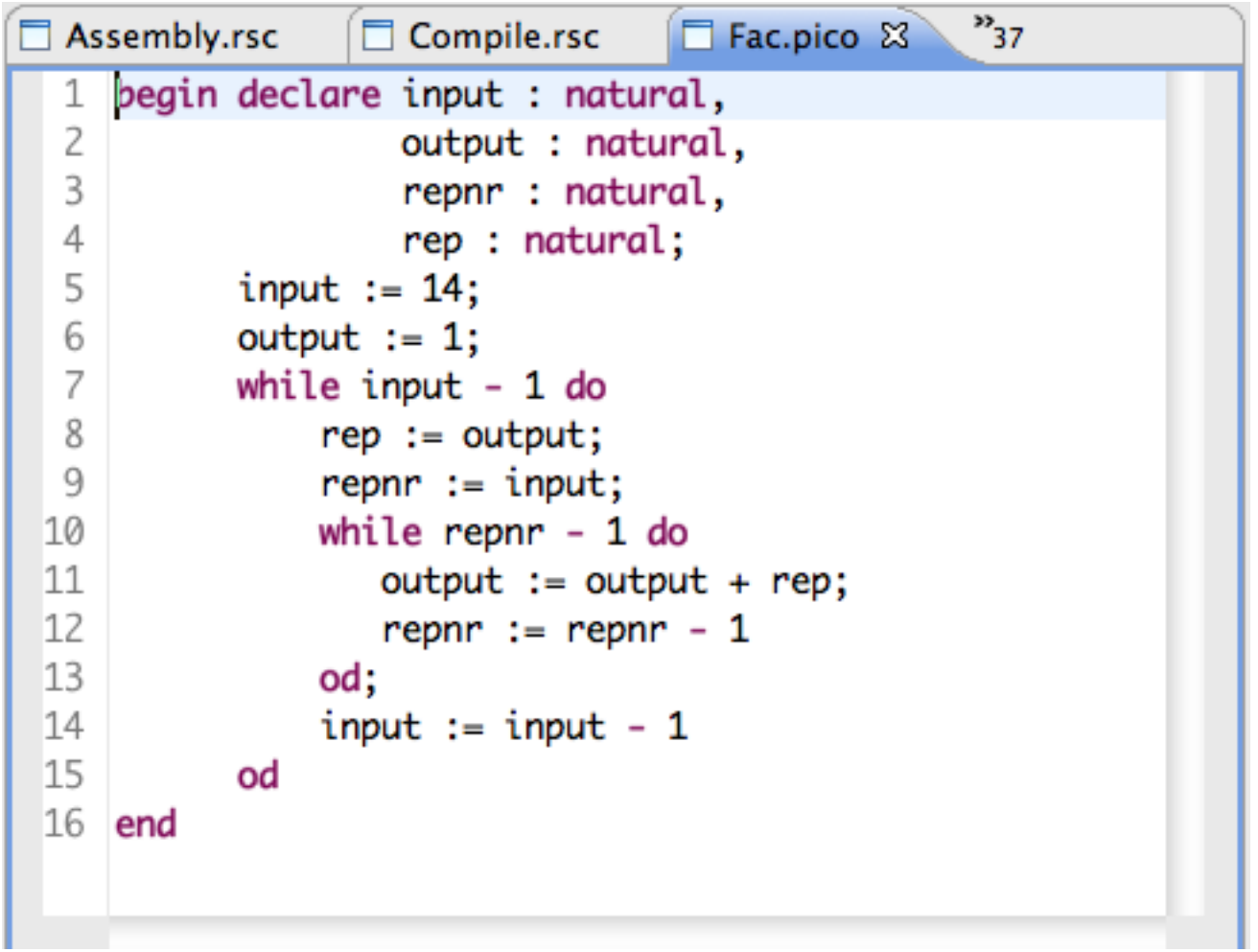
- There are two types: natural numbers and strings.
- Variables have to be declared.
- Statements are assignment, if-then-else, if-then and while-do.
- Expressions may contain naturals, strings, variables, addition (+), subtraction (-) and concatenation (||).
- The operators + and - have operands of type natural and their result is natural.
- The operator || has operands of type string and its results is also of type string.
- Tests in if-statement and while-statement should be of type natural (0 is false, ≠ 0 is true).

A Pico Program

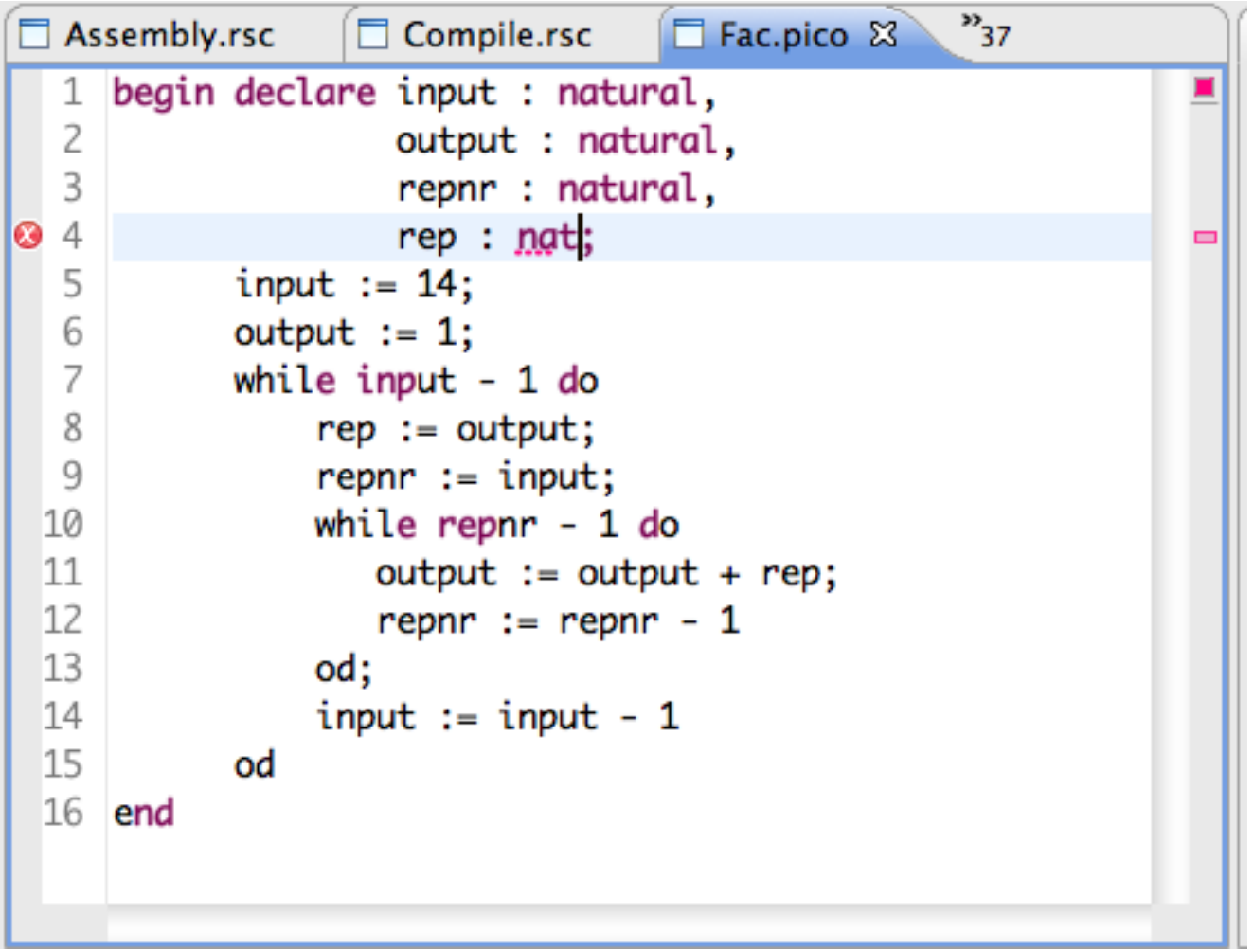
```
begin declare input : natural,
              output : natural,
              repnr : natural,
              rep : natural;
      input := 14;
      output := 1;
      while input - 1 do
          rep := output;
          repnr := input;
          while repnr - 1 do
             output := output + rep;
             repnr := repnr - 1
          od;
          input := input - 1
      od
end
```

- No input/output
- No multiplication
- What does this program do?

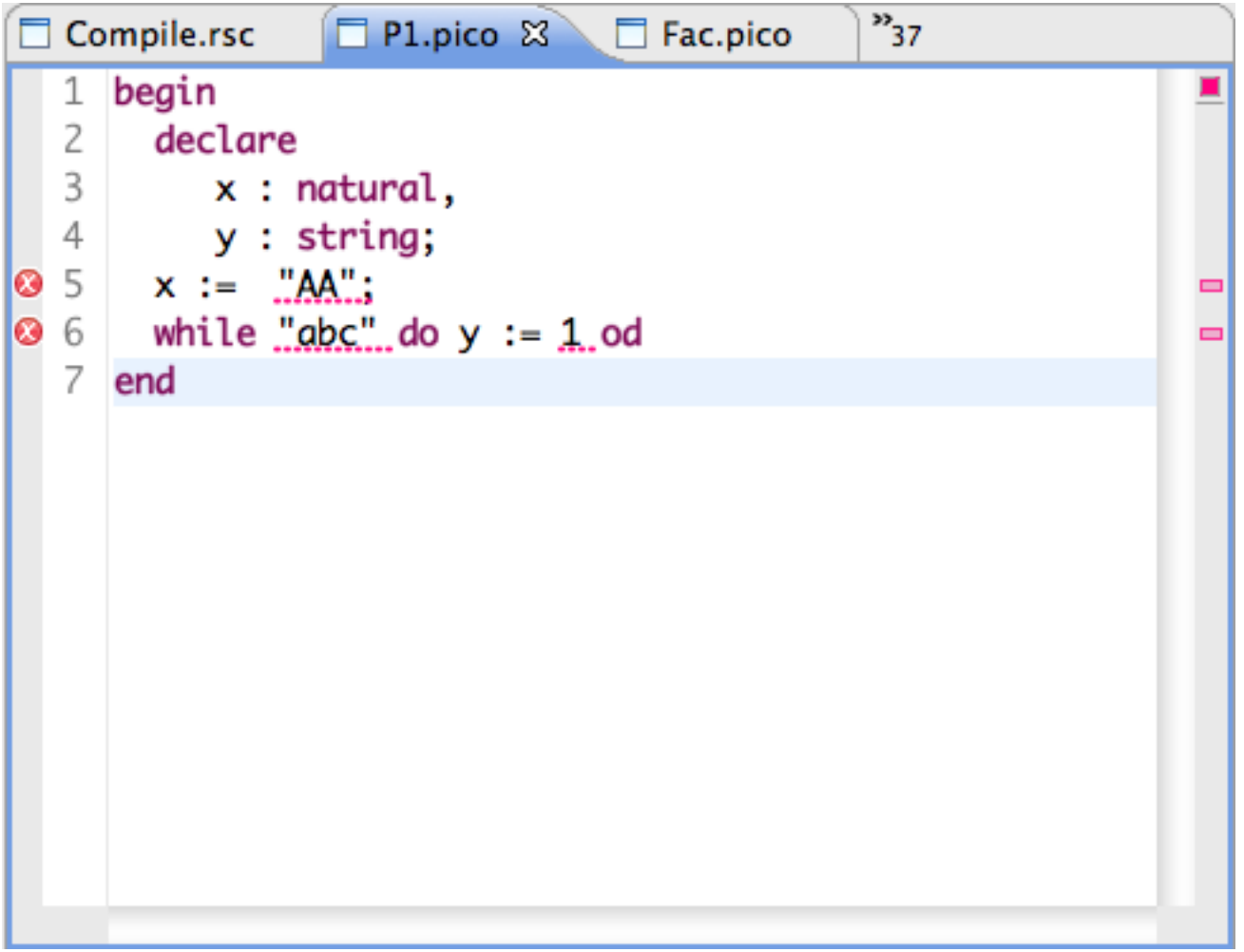
Parsing, Editing, Syntax highlighting



Signaling Parse Errors



Signaling Type Checking Errors



Compiling to Assembler

P1.pico | Fac.pico | »38

```
1  begin declare input : natural,
2                output : natural,
3                repnr : natural,
4                rep : natural;
5        input := 14;
6        output := 1;
7        while input - 1 do
8            rep := output;
9            repnr := input;
10           while repnr - 1 do
11               output := output + rep;
12               repnr := repnr - 1
13           od;
14           input := input - 1
15       od
16 end
```

Fac.asm

```
1  dclNat("input")
2  dclNat("output")
3  dclNat("repnr")
4  dclNat("rep")
5  lvalue("input")
6  pushNat(14)
7  assign()
8  lvalue("output")
9  pushNat(1)
10 assign()
11 label("L1")
12 rvalue("input")
13 pushNat(1)
14 sub2()
15 gofalse("L2")
16 lvalue("rep")
17 rvalue("output")
```

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

Pico Syntax, 1

```
module demo::lang::pico::Syntax

import Prelude;

lexical Id  = [a-z][a-z0-9]* !>> [a-z0-9];
lexical Natural = [0-9]+ ;
lexical String = "\"" ![\"]*  "\"";

layout Layout = WhitespaceAndComment* !>> [\ \t\n\r%];

lexical WhitespaceAndComment
   = [\ \t\n\r]
   | @category="Comment" "%" ![%]+ "%"
   | @category="Comment" "%%" ![\n]* $
   ;
```

Highlight as a comment

Pico Syntax, 2

```
start syntax Program
   = program: "begin" Declarations decls {Statement  ";"}* body "end" ;

syntax Declarations
   = "declare" {Declaration ","}* decls ";" ;

syntax Declaration = decl: Id id ":" Type tp;

syntax Type
   = natural:"natural"
   | string :"string"
   ;

syntax Statement
   = asgStat: Id var ":="  Expression val
   | ifElseStat: "if" Expression cond "then" {Statement ";"}*  thenPart
                                     "else" {Statement ";"}* elsePart "fi"
   | ifThenStat: "if" Expression cond "then" {Statement ";"}*  thenPart "fi"
   | whileStat: "while" Expression cond "do" {Statement ";"}* body "od"
  ;
```

Constructor names are added
to define the link with the abstract syntax

Pico Syntax, 3

```
syntax Expression
   = id: Id name
   | strCon: String string
   | natCon: Natural natcon
   | bracket "(" Expression e ")"
   > left conc: Expression lhs "||" Expression rhs
   > left ( add: Expression lhs "+" Expression rhs
          | sub: Expression lhs "-" Expression rhs
          )
  ;
```

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

Pico Abstract Syntax, 1

```
module demo::lang::pico::Abstract

public data TYPE =
  natural() | string();

public alias PicoId = str;

public data PROGRAM = program(list[DECL] decls, list[STATEMENT] stats);

public data DECL = decl(PicoId name, TYPE tp);

public data EXP =
      id(PicoId name)
    | natCon(int iVal)
    | strCon(str sVal)
    | add(EXP left, EXP right)
    | sub(EXP left, EXP right)
    | conc(EXP left, EXP right)
    ;

public data STATEMENT =
  asgStat(PicoId name, EXP exp)
| ifElseStat(EXP exp, list[STATEMENT] thenpart, list[STATEMENT] elsepart)
| ifThenStat(EXP exp, list[STATEMENT] thenpart)
| whileStat(EXP exp, list[STATEMENT] body)
;
```

Correspondence

Concrete Syntax		**Abstract Syntax**	
Program	program	PROGRAM	program
Declarations			
Declaration	decl	DECL	decl
Type	natural, string	TYPE	natural, string
Statement	asgStat, ifElseStat, ifThenStat, whileStat	STATEMENT	asgStat, ifElseStat, ifThenStat, whileStat
Expression	Id, strCon, natCon, conc, add, sub	EXP	Id, strCon, natCon, conc, add, sub

Pico Abstract Syntax, 2

```
anno loc TYPE@location;
anno loc PROGRAM@location;
anno loc DECL@location;
anno loc EXP@location;
anno loc STATEMENT@location;
```

For later convenience we also add declarations for annotations:

Read as:

- values of type `TYPE`, `PROGRAM`, ... can have an annotation
- The name of this annotation is `location`
- The type of this annotation is `loc`.

Usage:

- `implode` adds these annotations to the AST.
- Can be used in error message.
- Used by type checker

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

Load: Parse and make AST

```
module demo::lang::pico::Load

import Prelude;
import demo::lang::pico::Syntax;
import demo::lang::pico::Abstract;

public PROGRAM  load(str txt) = implode(#PROGRAM, parse(#Program, txt));
```

```
rascal>load("begin declare x : natural; x := 1 end");
PROGRAM: program(
  [decl(
      "x",
      natural()[
        @location=|file://-|(18,7,<1,18>,<1,25>)
      ])[
      @location=|file://-|(14,11,<1,14>,<1,25>)
    ]],
  [asgStat(
      "x",
      natCon(1)[
        @location=|file://-|(32,1,<1,32>,<1,33>)
      ])[
      @location=|file://-|(27,6,<1,27>,<1,33>)
    ]])[
  @location=|file://-|(0,37,<1,0>,<1,37>)
]
```

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

Pico Typechecking, 1

- Introduce type environment TENV consisting of
  - A mapping of identifiers and their type
  - A list of error messages so far

```
module demo::lang::pico::Typecheck

import Prelude;
import demo::lang::pico::Abstract;
import demo::lang::pico::Assembly;
import demo::lang::pico::Load;

alias TENV = tuple[ map[PicoId, TYPE] symbols, list[tuple[loc l, str msg]] errors];

TENV addError(TENV env, loc l, str msg) = env[errors = env.errors + <l, msg>];

str required(TYPE t, str got) = "Required <getName(t)>, got <got>";
str required(TYPE t1, TYPE t2) = required(t1, getName(t2));
```

Pico Typechecking, 2

- Define `TENV checkExp(EXP e, TYPE req, TENV env)`
  - `e` is the expression to be checked
  - `req` is its required type
  - `env` is the type environment.
- Returns:
  - If `e` has required type: `env`
  - If `e` does not have required type: `env` with an error message added to it.

Pico Typechecking, 3

checkExp for
- natCon
- strCon
- id

```
TENV checkExp(exp:natCon(int N), TYPE req, TENV env) =
  req == natural() ? env : addError(env, exp@location, required(req, "natural"));

TENV checkExp(exp:strCon(str S), TYPE req, TENV env) =
 req == string() ? env : addError(env, exp@location, required(req, "string"));

TENV checkExp(exp:id(PicoId Id), TYPE req, TENV env) {
  if(!env.symbols[Id]?)
     return addError(env, exp@location, "Undeclared variable <Id>");
  tpid = env.symbols[Id];
  return req == tpid ? env : addError(env, exp@location, required(req, tpid));
}
```

Retrieve the location annotation

Pico Typechecking, 4

checkExp for

- add
- sub
- conc

```
TENV checkExp(exp:add(EXP E1, EXP E2), TYPE req, TENV env) =
  req == natural() ? checkExp(E1, natural(), checkExp(E2, natural(), env))
                   : addError(env, exp@location, required(req, "natural"));

TENV checkExp(exp:sub(EXP E1, EXP E2), TYPE req, TENV env) =
  req == natural() ? checkExp(E1, natural(), checkExp(E2, natural(), env))
                   : addError(env, exp@location, required(req, "natural"));

TENV checkExp(exp:conc(EXP E1, EXP E2), TYPE req, TENV env) =
  req == string() ? checkExp(E1, string(), checkExp(E2, string(), env))
                  : addError(env, exp@location, required(req, "string"));
```

Pico Typechecking, 5

```
TENV checkStat(stat:asgStat(PicoId Id, EXP Exp), TENV env) {
  if(!env.symbols[Id]?)
     return addError(env, stat@location, "Undeclared variable <Id>");
  tpid = env.symbols[Id];
  return checkExp(Exp, tpid, env);
}

TENV checkStat(stat:ifElseStat(EXP Exp,
                               list[STATEMENT] Stats1,
                               list[STATEMENT] Stats2),
               TENV env){
    env0 = checkExp(Exp, natural(), env);
    env1 = checkStats(Stats1, env0);
    env2 = checkStats(Stats2, env1);
    return env2;
}

TENV checkStat(stat:ifThenStat(EXP Exp, list[STATEMENT] Stats1), TENV env){...}

TENV checkStat(stat:whileStat(EXP Exp, list[STATEMENT] Stats1), TENV env) {...}
```

Pico Typechecking, 6

```
TENV checkStats(list[STATEMENT] Stats1, TENV env) {
  for(S <- Stats1){
      env = checkStat(S, env);
  }
  return env;
}
```

Pico Typechecking, 7

```
TENV checkDecls(list[DECL] Decls) =
    <( Id : tp  | decl(PicoId Id, TYPE tp) <- Decls), []>;
```

Pico Typechecking, 8

```
public TENV checkProgram(PROGRAM P){
  if(program(list[DECL] Decls, list[STATEMENT] Series) := P){
     TENV env = checkDecls(Decls);
     return checkStats(Series, env);
  } else
    throw "Cannot happen";
}
```

Pico Typechecking, 9

```
public list[tuple[loc l, str msg]] checkProgram(str txt) =
       checkProgram(load(txt)).errors;
```

```
rascal>checkProgram("begin declare x : natural; x := 1 end")
list[tuple[loc l,str msg]]: []


rascal>checkProgram("begin declare x : string; x := 1 end")
list[tuple[loc l,str msg]]: [<|file://-|(31,1,<1,31>,<1,32>),
                              "Required string, got natural">]


rascal>checkProgram("begin declare x : string; z := 1 end")
list[tuple[loc l,str msg]]: [<|file://-|(26,6,<1,26>,<1,32>),
                             "Undeclared variable z">]
```

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

Assembly Language

```
module demo::lang::pico::Assembly

import demo::lang::pico::Abstract;

public data Instr =
  dclNat(PicoId Id)  | dclStr(PicoId Id)
| pushNat(int intCon)| pushStr(str strCon)
| rvalue(PicoId Id)
| lvalue(PicoId Id)
| pop() | copy()
| assign()
| add2() | sub2() | conc2()
| label(str label) | go(str  label)
| gotrue(str label) | gofalse(str label);
```

Reserve a location for a variable

Push int/str value

rvalue: push value of variable

lvalue: push address of variable

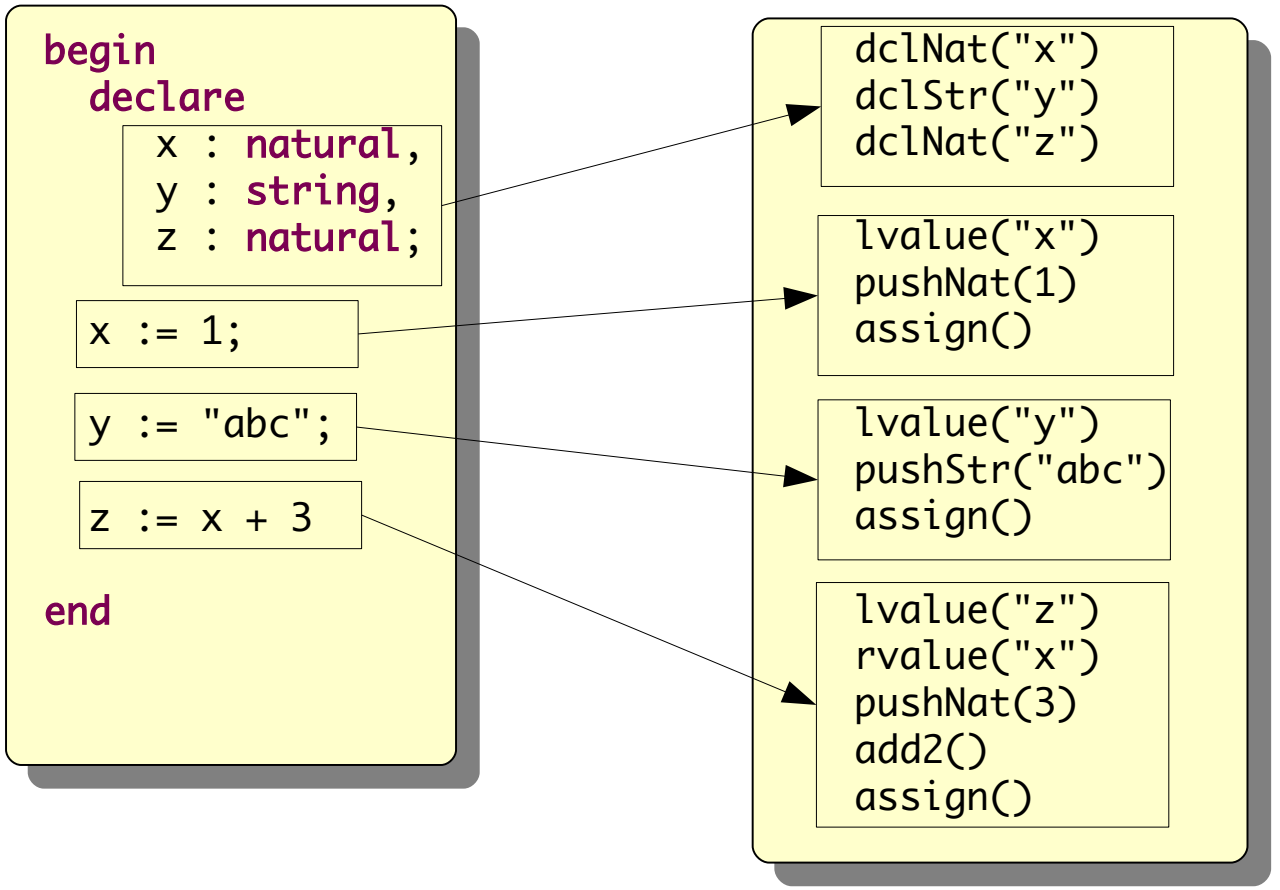
assign: given address of variable and value assign value to it.

Expression operators

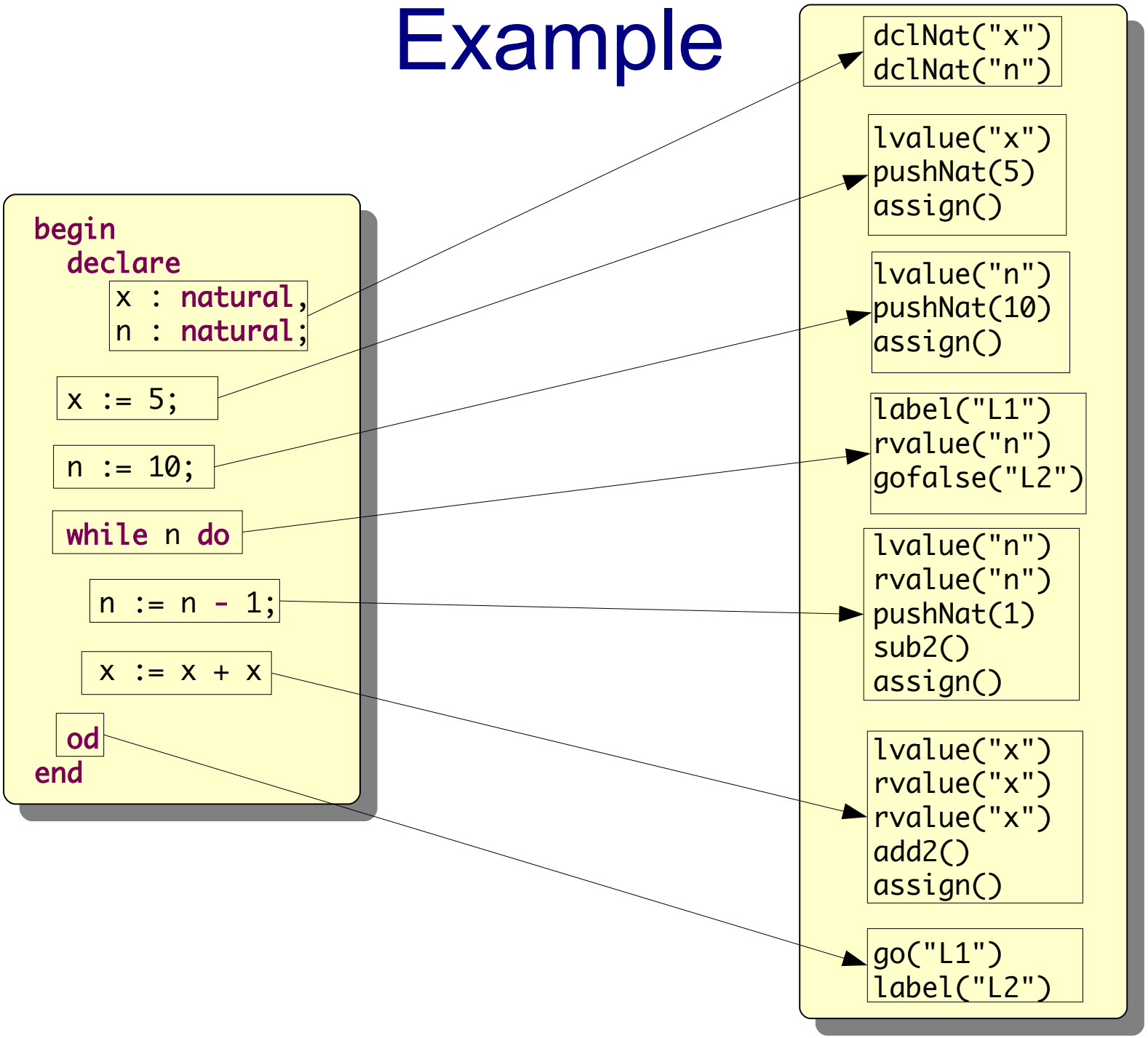
Labels and goto's

A stack-based machine

Example


begin
declare
x : natural,
y : string,
z : natural;
x := 1;
y := "abc";
z := x + 3
end
dclNat("x")
dclStr("y")
dclNat("z")
lvalue("x")
pushNat(1)
assign()
lvalue("y")
pushStr("abc")
assign()
lvalue("z")
rvalue("x")
pushNat(3)
add2()
assign()

Example


begin
declare
x : natural,
n : natural;
x := 5;
n := 10;
while n do
n := n - 1;
x := x + x
od
end
dclNat("x")
dclNat("n")
lvalue("x")
pushNat(5)
assign()
lvalue("n")
pushNat(10)
assign()
label("L1")
rvalue("n")
gofalse("L2")
lvalue("n")
rvalue("n")
pushNat(1)
sub2()
assign()
lvalue("x")
rvalue("x")
rvalue("x")
add2()
assign()
go("L1")
label("L2")

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

Compile Expressions

```
alias Instrs = list[Instr];

// compile Expressions.

Instrs compileExp(natCon(int N)) = [pushNat(N)];

Instrs compileExp(strCon(str S)) = [pushStr(substring(S,1,size(S)-1))];

Instrs compileExp(id(PicoId Id)) = [rvalue(Id)];

public Instrs compileExp(add(EXP E1, EXP E2)) =
  [*compileExp(E1), *compileExp(E2), add2()];

Instrs compileExp(sub(EXP E1, EXP E2)) =
  [*compileExp(E1), *compileExp(E2), sub2()];

Instrs compileExp(conc(EXP E1, EXP E2)) =
  [*compileExp(E1), *compileExp(E2), conc2()];
```

Strip surrounding string quotes

Unique Label Generation

```
// Unique label generation

private int nLabel = 0;

private str nextLabel() {
  nLabel += 1;
  return "L<nLabel>";
}
```

- Generates: L1, L2, L3, ...
- Used for compiling if statements and while loop

Compile Statement: Assignment

```
Instrs compileStat(asgStat(PicoId Id, EXP Exp)) =
    [lvalue(Id), *compileExp(Exp), assign()];
```

Compile Statement: IfElse

```
Instrs compileStat(ifElseStat(EXP Exp,
                              list[STATEMENT] Stats1,
                              list[STATEMENT] Stats2)){
  nextLab = nextLabel();
  falseLab = nextLabel();
  return [*compileExp(Exp),
          gofalse(falseLab),
          *compileStats(Stats1),
          go(nextLab),
          label(falseLab),
          *compileStats(Stats2),
          label(nextLab)];
}
```

Compile Statement: While

```
Instrs compileStat(whileStat(EXP Exp,
                             list[STATEMENT] Stats1)) {
  entryLab = nextLabel();
  nextLab = nextLabel();
  return [label(entryLab),
          *compileExp(Exp),
          gofalse(nextLab),
          *compileStats(Stats1),
          go(entryLab),
          label(nextLab)];
}
```

Compile Statements

```
Instrs compileStats(list[STATEMENT] Stats1) =
  [ *compileStat(S) | S <- Stats1 ];
```

Compile Declarations

```
Instrs compileDecls(list[DECL] Decls) =
  [ ((tp == natural()) ? dclNat(Id) : dclStr(Id))  |
    decl(PicoId Id, TYPE tp) <- Decls
  ];
```

Compile a Pico Program

```
public Instrs compileProgram(PROGRAM P){
  nLabel = 0;
  if(program(list[DECL] Decls, list[STATEMENT] Series) := P){
     return [*compileDecls(Decls), *compileStats(Series)];
  } else
    throw "Cannot happen";
}
```

The Final Pico -> ASM Compiler

```
public Instrs compileProgram(str txt) = compileProgram(load(txt));
```

```
rascal>compileProgram("begin declare x : natural; x := 1 end")
Instrs: [
  dclNat("x"),
  lvalue("x"),
  pushNat(1),
  assign()
]
```

Plan for Pico

- Define Concrete Syntax
- Define Abstract Syntax
- Define translation Parse Tree -> AST
- Define Type Checker
- Define Assembly Language ASM
- Define Compiler Pico -> ASM
- Integrate all these tools in Eclipse

The Pico Plugin

- Register the Pico language in Eclipse:
  - Name of the language: Pico
  - File name extension: pico
  - Wrap parser, checker and compiler for use from Eclipse
  - Define the “contributions” to the Eclipse GUI (menu entries, buttons, ...)
  - Register all tools to work on Pico files

Pico Plugin: Preliminaries

```
module demo::lang::pico::Pico

import Prelude;
import util::IDE;
import demo::lang::pico::Abstract;
import demo::lang::pico::Syntax;
import demo::lang::pico::Typecheck;
import demo::lang::pico::Compile;


private str Pico_NAME = "Pico";
private str Pico_EXT = "pico";
```

Wrap parser and Typechecker

```
// Parsing
Tree parser(str x, loc l) {
    return parse(#demo::lang::pico::Syntax::Program, x, l);
}

// Type checking

public Program checkPicoProgram(Program x) {
p = implode(#PROGRAM, x);
env = checkProgram(p);
errors = { error(s, l) | <l, s> <- env.errors };
return x[@messages = errors];
}
```

Current Parse Tree in editor

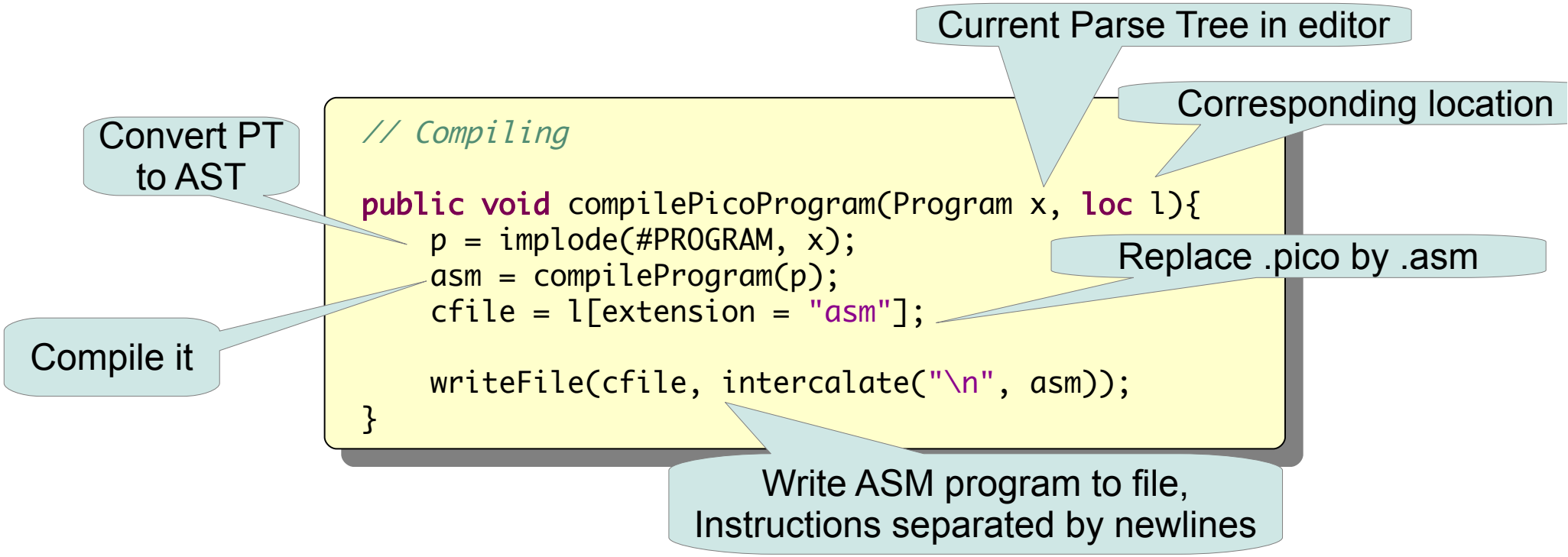
Convert PT to AST

Make a list of error messages

Typecheck it

Return original tree with error messages in messages annotation

Wrap Compiler




```
// Compiling

public void compilePicoProgram(Program x, loc l){
    p = implode(#PROGRAM, x);
    asm = compileProgram(p);
    cfile = l[extension = "asm"];

    writeFile(cfile, intercalate("\n", asm));
}
```

IDE Contributions

```
// Contributions to IDE

public set[Contribution] Pico_CONTRIBS = {
popup(
menu("Pico",[
          action("Compile Pico to ASM", compilePicoProgram)
    ])
    )
};
```

Define menu entry “Pico” with
Submenu item “Compile Pico to ASM”

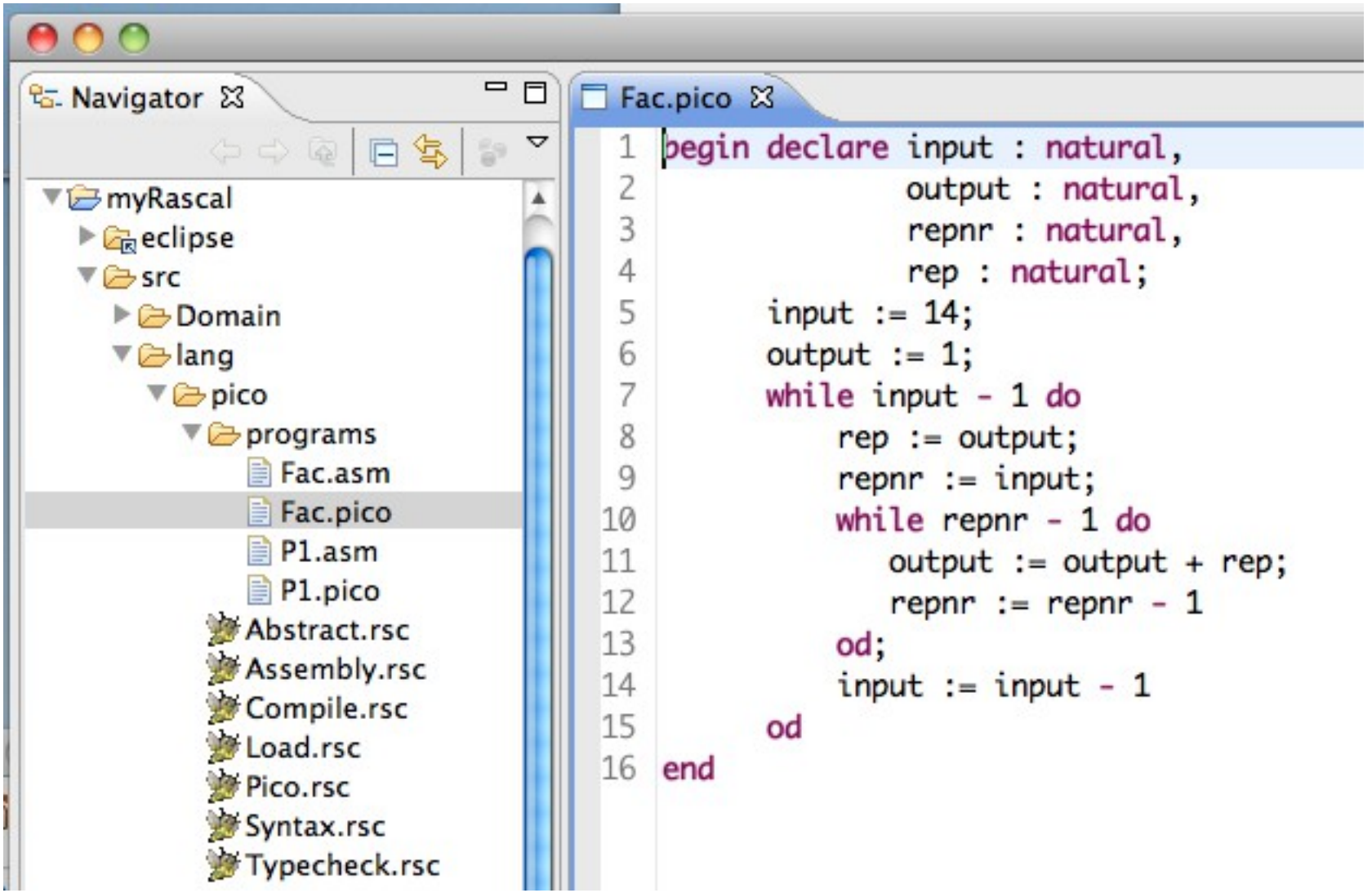
Register Pico

```
// Register the Pico tools

public void registerPico() {
  registerLanguage(Pico_NAME, Pico_EXT, parser);
  registerAnnotator(Pico_NAME, checkPicoProgram);
  registerContributions(Pico_NAME, Pico_CONTRIBS);
}
```

```
rascal>import demo::lang::pico::Pico;
ok

rascal>registerPico();
ok
```

Clicking on a `.pico` file will now open a Pico editor

Opening a Pico Editor



Further Reading

- Tutor: Recipes/Languages
- Tutor: Rascal/Libraries/util/IDE