PPA Final Presentation: Saulang

Static analyser for toy programming language

Team members

- *Saul Goodman*
- Egor Shalagin
- Mikhail Fedorov
- Davlatkhodzha Magzumov

Our goals

- Write Lexer and Parser for a toy functional language
- Write Type Checker for Static Analysis of our language
- Provide tests for Type Checking, and check our program on them
- Write simple documentation on how to use our program.

Language

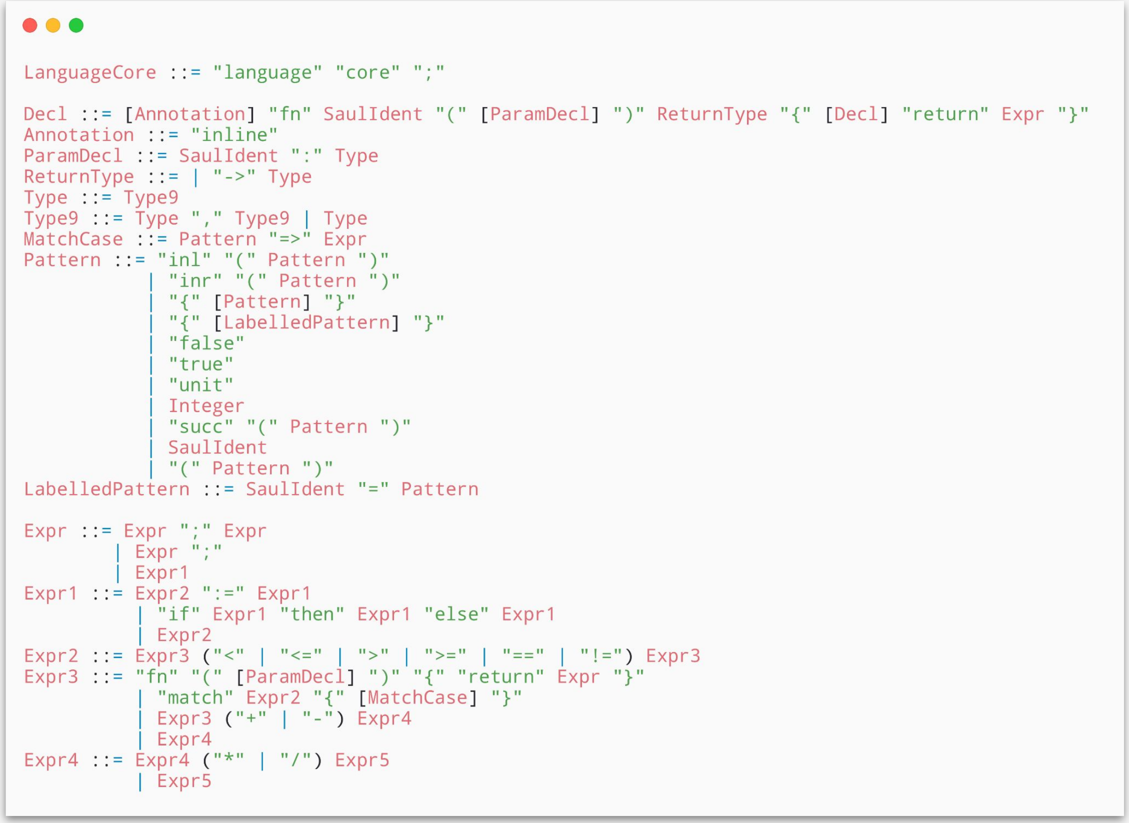
Type: function programming language

Typing system: static

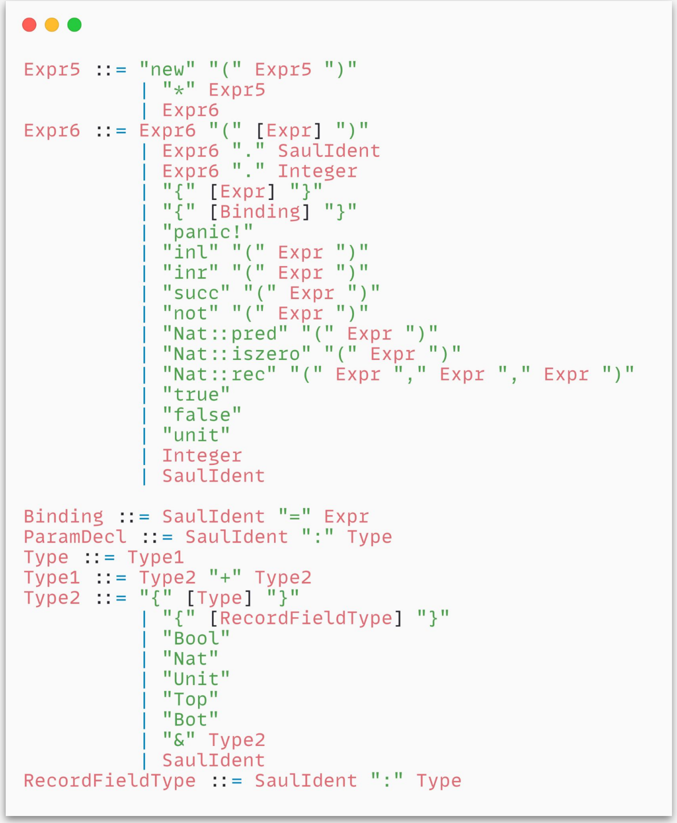
Containers: tuples, variables, records (structures),

Features: language core and some interesting constructs (such as, sum types, subtyping, pattern matching, ect)

Language formal grammar

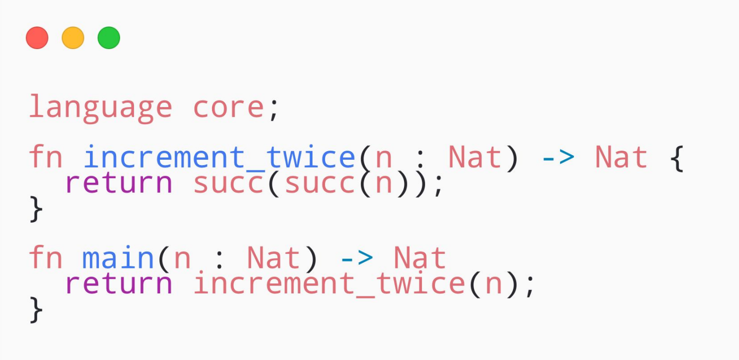


```
LanguageCore ::= "language" "core" ";"

Decl ::= [Annotation] "fn" SaulIdent "(" [ParamDecl] ")" ReturnType "{" [Decl] "return" Expr "}"
Annotation ::= "inline"
ParamDecl ::= SaulIdent ":" Type
ReturnType ::= | "->" Type
Type ::= Type9
Type9 ::= Type "," Type9 | Type
MatchCase ::= Pattern "=>" Expr
Pattern ::= "inl" "(" Pattern ")"
           | "inr" "(" Pattern ")"
           | "{" [Pattern] "}"
           | "{" [LabelledPattern] "}"
           | "false"
           | "true"
           | "unit"
           | Integer
           | "succ" "(" Pattern ")"
           | SaulIdent
           | "(" Pattern ")"
LabelledPattern ::= SaulIdent "=" Pattern

Expr ::= Expr ";" Expr
        | Expr ";"
        | Expr1
Expr1 ::= Expr2 ":=" Expr1
          | "if" Expr1 "then" Expr1 "else" Expr1
          | Expr2
Expr2 ::= Expr3 ("<" | "<=" | ">" | ">=" | "==" | "!=") Expr3
Expr3 ::= "fn" "(" [ParamDecl] ")" "{" "return" Expr "}"
          | "match" Expr2 "{" [MatchCase] "}"
          | Expr3 ("+" | "-") Expr4
          | Expr4
Expr4 ::= Expr4 ("*" | "/") Expr5
          | Expr5
```

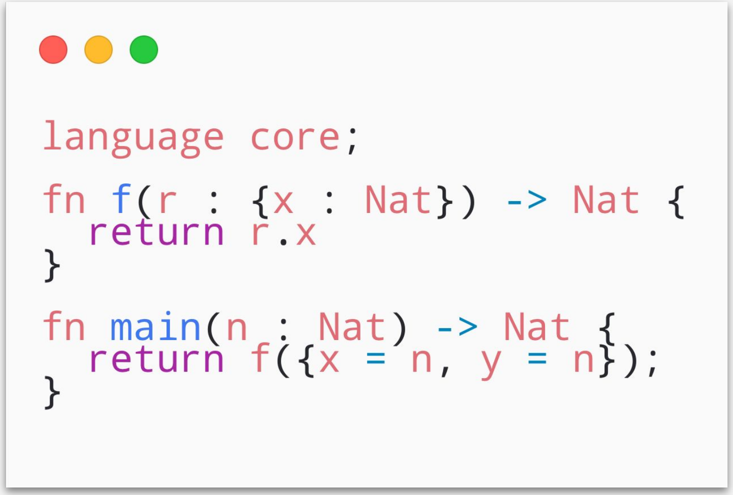


```
Expr5 ::= "new" "(" Expr5 ")"
          | "*" Expr5
          | Expr6
Expr6 ::= Expr6 "(" [Expr] ")"
          | Expr6 "." SaulIdent
          | Expr6 "." Integer
          | "{" [Expr] "}"
          | "{" [Binding] "}"
          | "panic!"
          | "inl" "(" Expr ")"
          | "inr" "(" Expr ")"
          | "succ" "(" Expr ")"
          | "not" "(" Expr ")"
          | "Nat::pred" "(" Expr ")"
          | "Nat::iszero" "(" Expr ")"
          | "Nat::rec" "(" Expr "," Expr "," Expr ")"
          | "true"
          | "false"
          | "unit"
          | Integer
          | SaulIdent

Binding ::= SaulIdent "=" Expr
ParamDecl ::= SaulIdent ":" Type
Type ::= Type1
Type1 ::= Type2 "+" Type2
Type2 ::= "{" [Type] "}"
          | "{" [RecordFieldType] "}"
          | "Bool"
          | "Nat"
          | "Unit"
          | "Top"
          | "Bot"
          | "&" Type2
          | SaulIdent
RecordFieldType ::= SaulIdent ":" Type
```

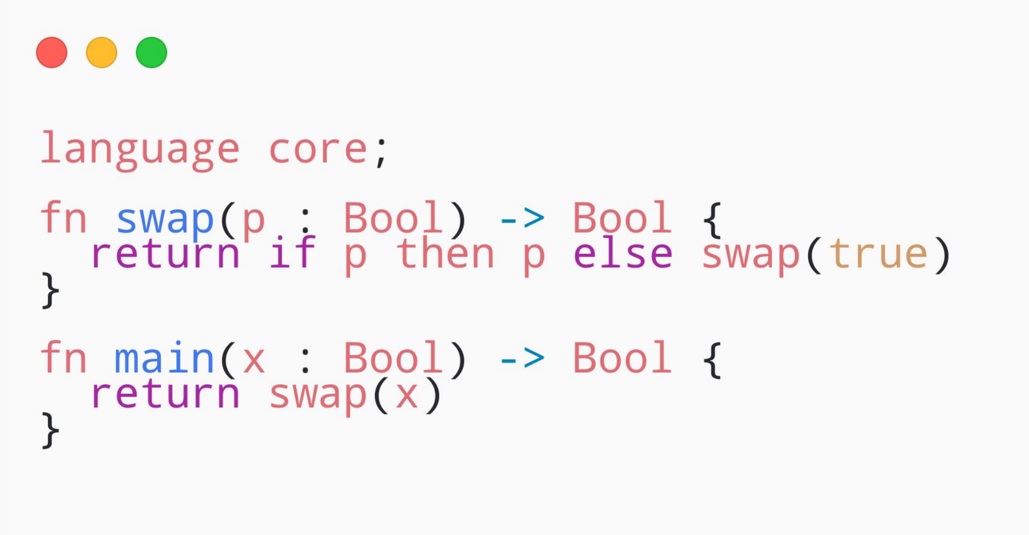
Language examples



```
language core;

fn increment_twice(n : Nat) -> Nat {
  return succ(succ(n));
}

fn main(n : Nat) -> Nat
  return increment_twice(n);
}
```



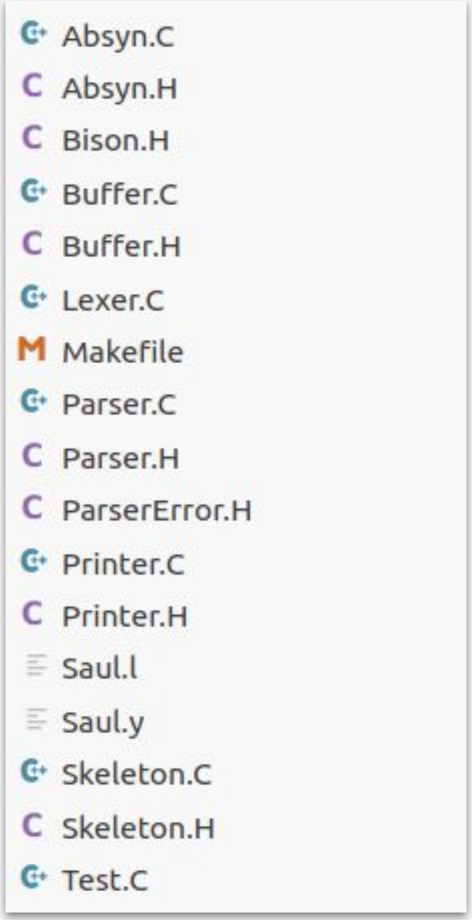
```
language core;

fn f(r : {x : Nat}) -> Nat {
  return r.x
}

fn main(n : Nat) -> Nat {
  return f({x = n, y = n});
}
```



```
language core;

fn swap(p : Bool) -> Bool {
  return if p then p else swap(true)
}

fn main(x : Bool) -> Bool {
  return swap(x)
}
```

Lexer and Parser

- Lexer and Parser were generated by BNFC
- Formal grammar was rewritten to Backus-Naur Form
- command **bnfc -m --cpp ../../Saul.cf -l -p Saul** to generate files



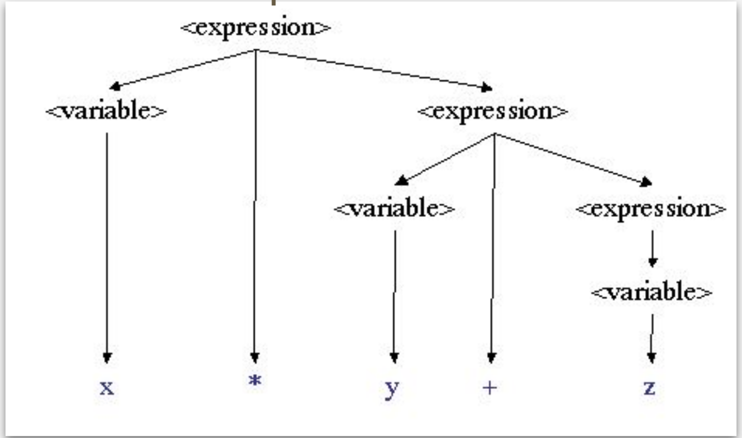
```
Absyn.C
Absyn.H
Bison.H
Buffer.C
Buffer.H
Lexer.C
Makefile
Parser.C
Parser.H
ParserError.H
Printer.C
Printer.H
Saul.l
Saul.y
Skeleton.C
Skeleton.H
Test.C
```

What is BNF Converter?

BNF Converter is a tool that allows you to generate parsers and [illegible] work with [illegible] AST [illegible] is based on describing the syntax of programming languages. [illegible] a formalism for

What is BNF?

Backus-Naur Form (BNF) is a meta-language used to describe the syntax of programming languages, command languages, and document formats. It is a notation for context-free grammars, which are used to describe the structure of languages. BNF is named after John Backus and Peter Naur, who developed it in the 1950s as part of the development of the ALGOL programming language.



How does BNFC work?

BNFC takes as input a BNF grammar

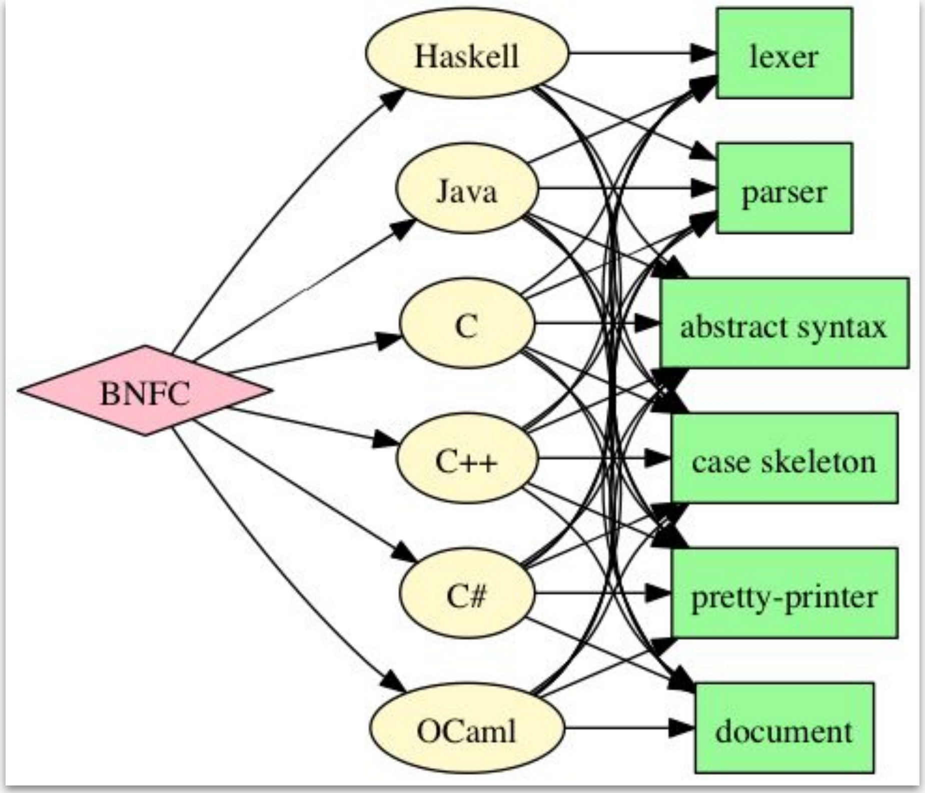
produces source code for a parser and

language defined by the grammar. the

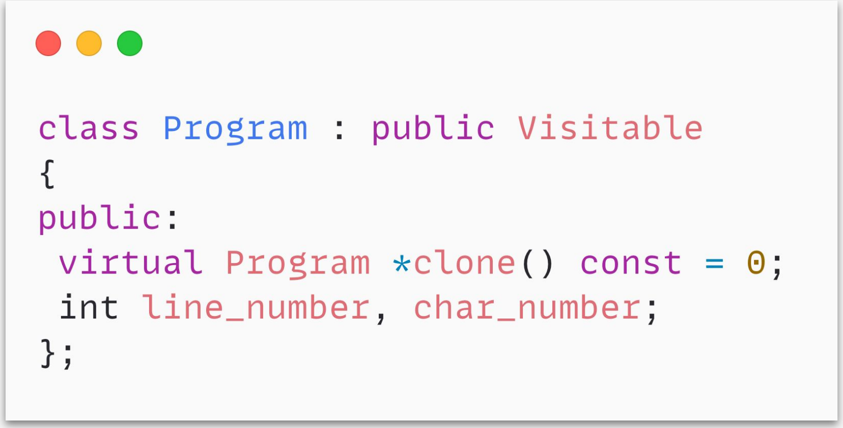
The tool supports a wide range of

programming languages, including C,

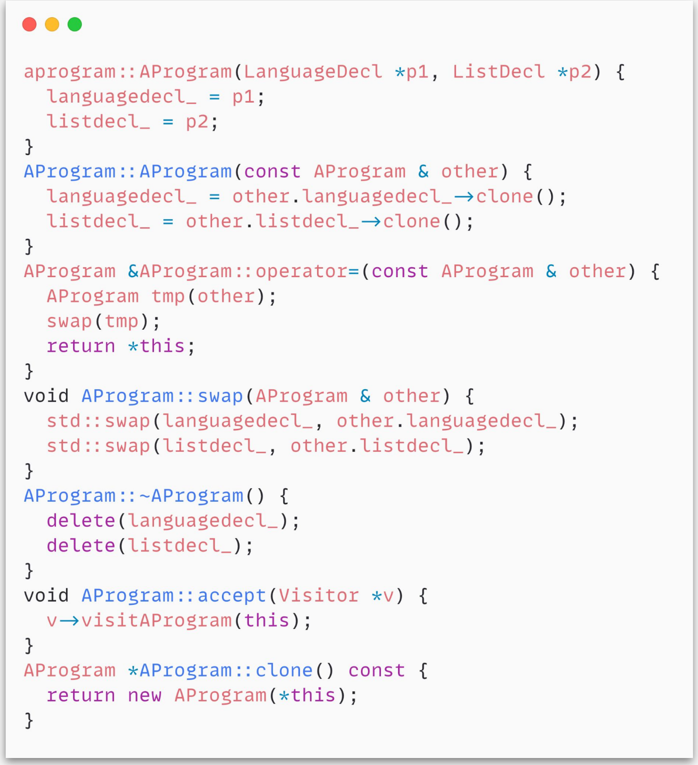
Java, Python, and many others.



BNFC Tree nodes

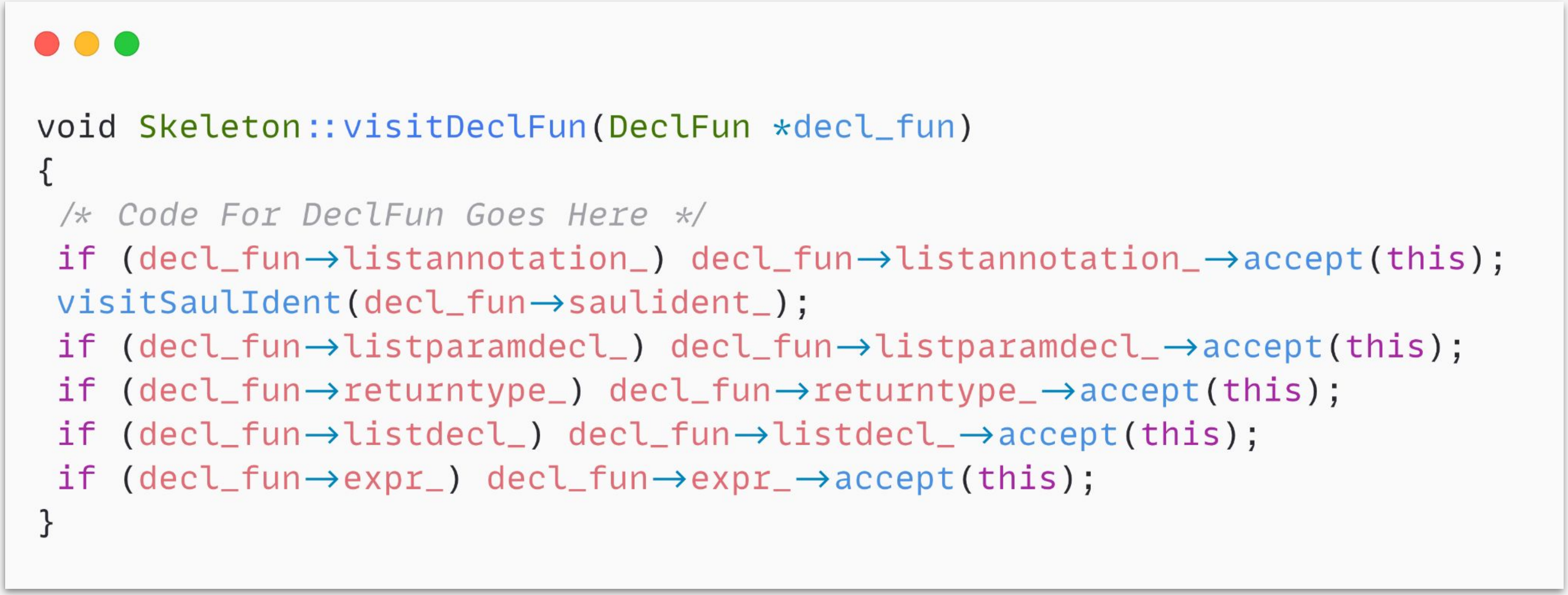


```
class Program : public Visitable
{
public:
 virtual Program *clone() const = 0;
 int line_number, char_number;
};
```



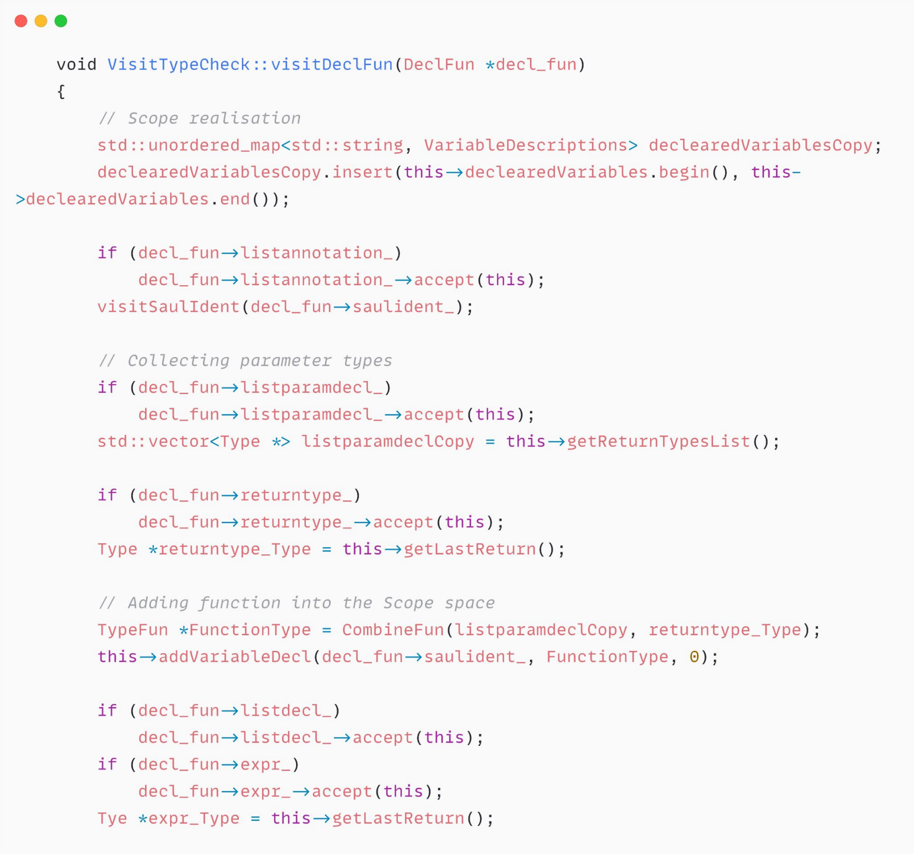
```
aprogram::AProgram(LanguageDecl *p1, ListDecl *p2) {
  languagedecl_ = p1;
  listdecl_ = p2;
}
AProgram::AProgram(const AProgram & other) {
  languagedecl_ = other.languagedecl_->clone();
  listdecl_ = other.listdecl_->clone();
}
AProgram &AProgram::operator=(const AProgram & other) {
  AProgram tmp(other);
  swap(tmp);
  return *this;
}
void AProgram::swap(AProgram & other) {
  std::swap(languagedecl_, other.languagedecl_);
  std::swap(listdecl_, other.listdecl_);
}
AProgram::~AProgram() {
  delete(languagedecl_);
  delete(listdecl_);
}
void AProgram::accept(Visitor *v) {
  v->visitAProgram(this);
}
AProgram *AProgram::clone() const {
  return new AProgram(*this);
}
```

Visitor pattern: BNFC generated

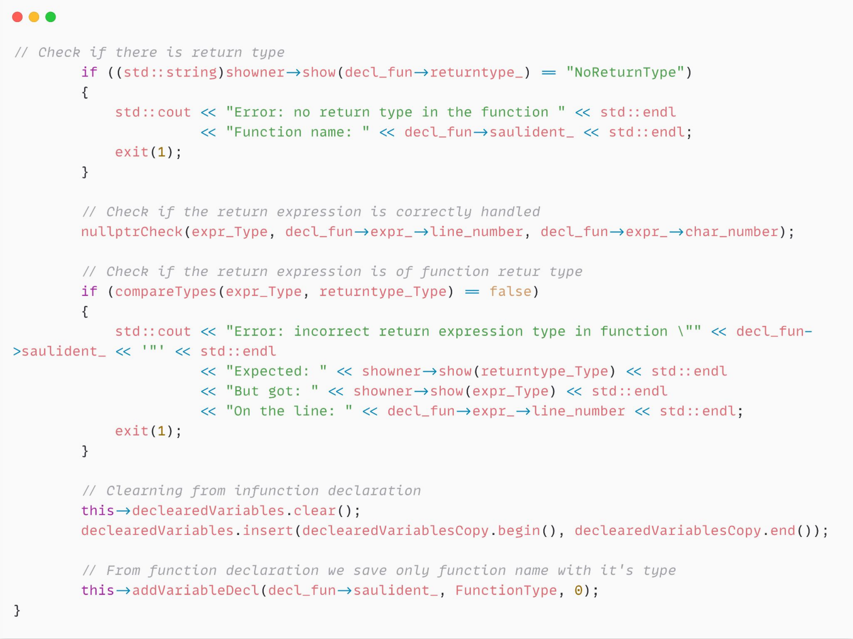


```
void Skeleton::visitDeclFun(DeclFun *decl_fun)
{
 /* Code For DeclFun Goes Here */
 if (decl_fun->listannotation_) decl_fun->listannotation_->accept(this);
 visitSaulIdent(decl_fun->saulident_);
 if (decl_fun->listparamdecl_) decl_fun->listparamdecl_->accept(this);
 if (decl_fun->returntype_) decl_fun->returntype_->accept(this);
 if (decl_fun->listdecl_) decl_fun->listdecl_->accept(this);
 if (decl_fun->expr_) decl_fun->expr_->accept(this);
}
```

Visitor pattern: final function

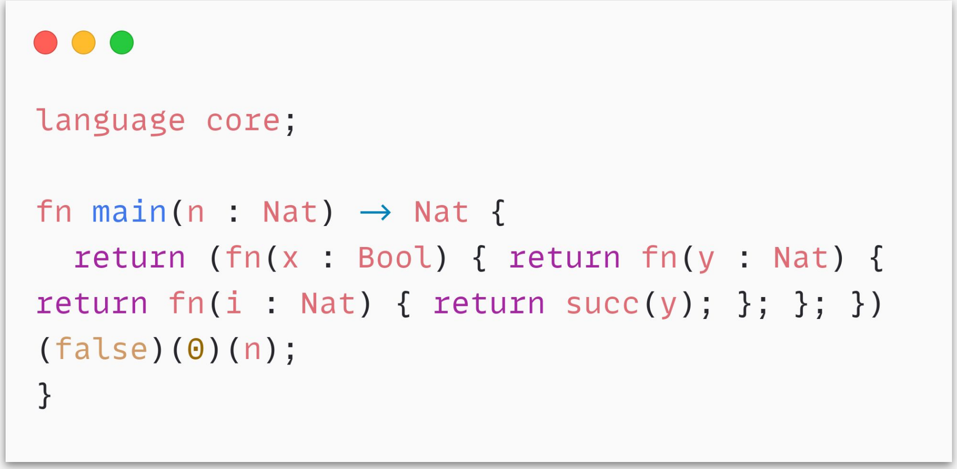


```
    void VisitTypeCheck::visitDeclFun(DeclFun *decl_fun)
    {
        // Scope realisation
        std::unordered_map<std::string, VariableDescriptions> decleardVariablesCopy;
        decleardVariablesCopy.insert(this->decleardVariables.begin(), this-
>decleardVariables.end());

        if (decl_fun->listannotation_)
            decl_fun->listannotation_->accept(this);
        visitSaulIdent(decl_fun->saulident_);

        // Collecting parameter types
        if (decl_fun->listparamdecl_)
            decl_fun->listparamdecl_->accept(this);
        std::vector<Type *> listparamdeclCopy = this->getReturnTypesList();

        if (decl_fun->returntype_)
            decl_fun->returntype_->accept(this);
        Type *returntype_Type = this->getLastReturn();

        // Adding function into the Scope space
        TypeFun *FunctionType = CombineFun(listparamdeclCopy, returntype_Type);
        this->addVariableDecl(decl_fun->saulident_, FunctionType, 0);

        if (decl_fun->listdecl_)
            decl_fun->listdecl_->accept(this);
        if (decl_fun->expr_)
            decl_fun->expr_->accept(this);
        Tye *expr_Type = this->getLastReturn();
```

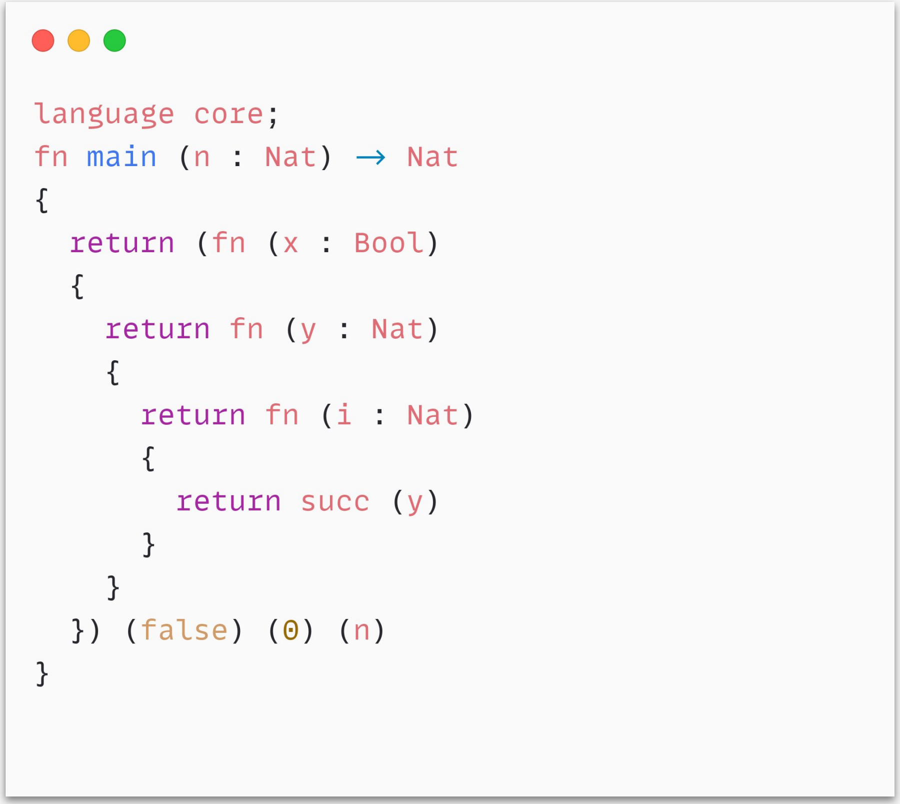


```
// Check if there is return type
        if ((std::string)showner->show(decl_fun->returntype_) == "NoReturnType")
        {
            std::cout << "Error: no return type in the function " << std::endl
                      << "Function name: " << decl_fun->saulident_ << std::endl;
            exit(1);
        }

        // Check if the return expression is correctly handled
        nullptrCheck(expr_Type, decl_fun->expr_->line_number, decl_fun->expr_->char_number);

        // Check if the return expression is of function retur type
        if (compareTypes(expr_Type, returntype_Type) == false)
        {
            std::cout << "Error: incorrect return expression type in function \"" << decl_fun-
>saulident_ << '"' << std::endl
                      << "Expected: " << showner->show(returntype_Type) << std::endl
                      << "But got: " << showner->show(expr_Type) << std::endl
                      << "On the line: " << decl_fun->expr_->line_number << std::endl;
            exit(1);
        }

        // Clearning from infunction declaration
        this->decleardVariables.clear();
        decleardVariables.insert(decleardVariablesCopy.begin(), decleardVariablesCopy.end());

        // From function declaration we save only function name with it's type
        this->addVariableDecl(decl_fun->saulident_, FunctionType, 0);
}
```

BNFC: code automatic code formatting

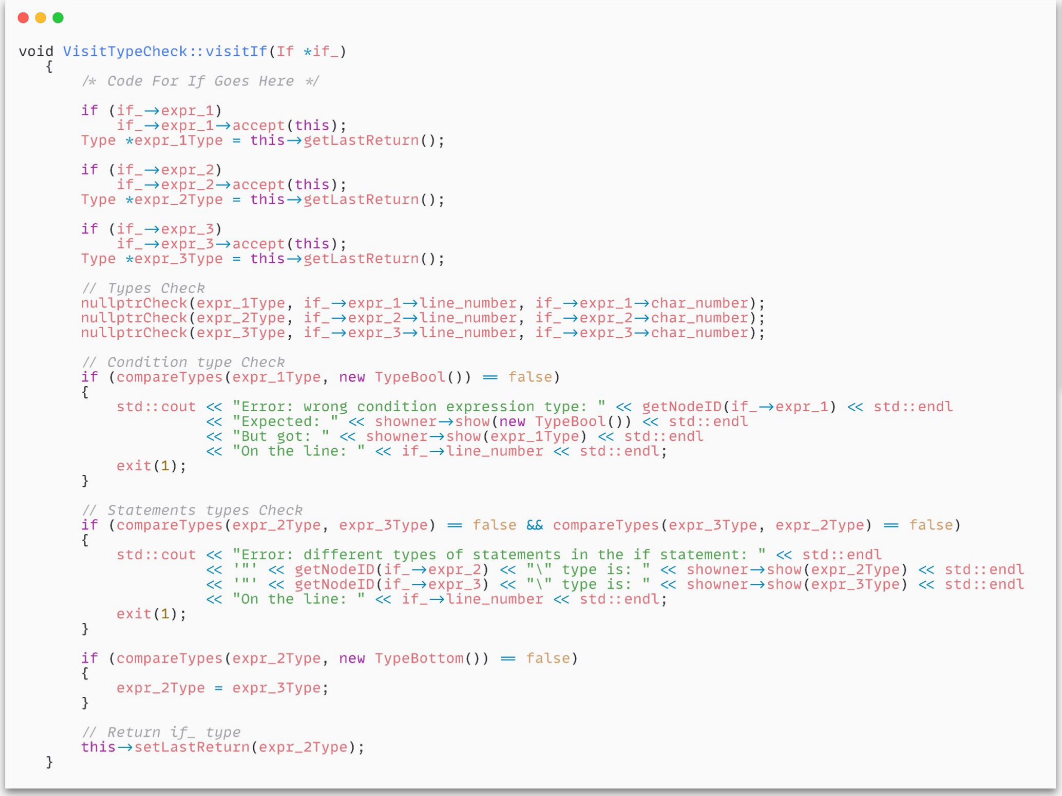


```
language core;

fn main(n : Nat) → Nat {
  return (fn(x : Bool) { return fn(y : Nat) {
return fn(i : Nat) { return succ(y); }; }; })
(false)(0)(n);
}
```

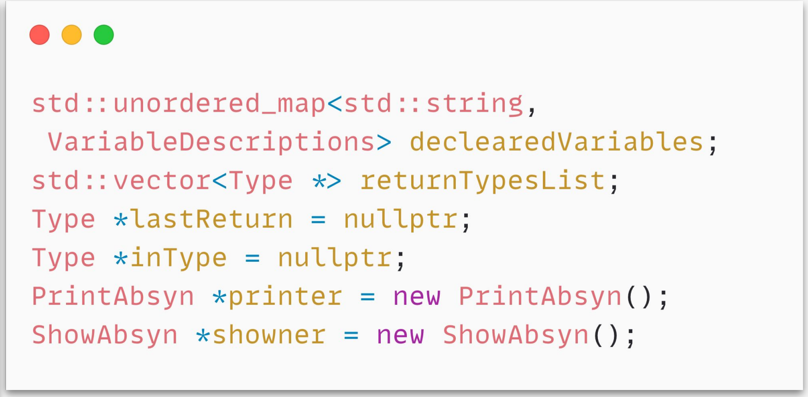


```
language core;
fn main (n : Nat) → Nat
{
  return (fn (x : Bool)
  {
    return fn (y : Nat)
    {
      return fn (i : Nat)
      {
        return succ (y)
      }
    }
  }) (false) (0) (n)
}
```

Semantic Analysis: implementation example

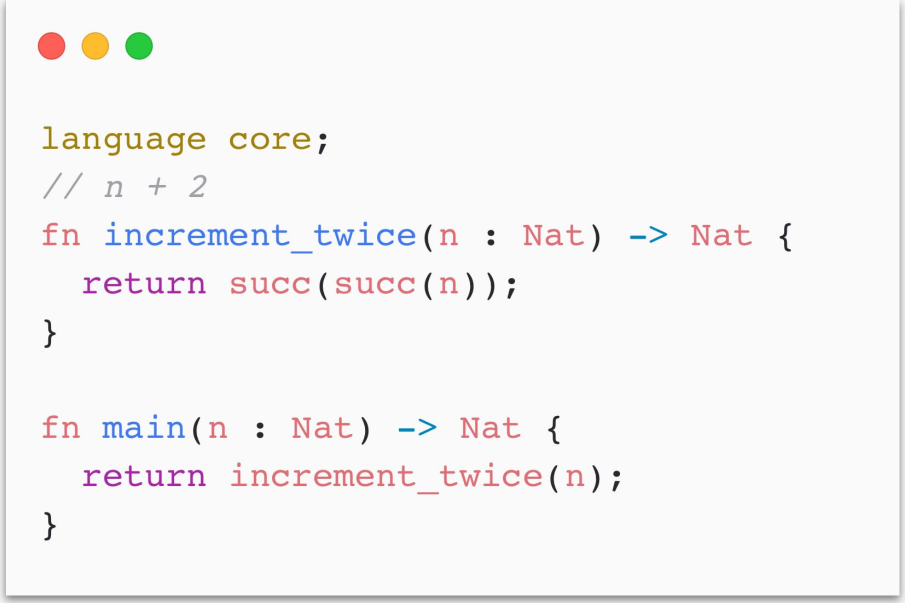


```
void VisitTypeCheck::visitIf(If *if_)
    {
        /* Code For If Goes Here */

        if (if_->expr_1)
            if_->expr_1->accept(this);
        Type *expr_1Type = this->getLastReturn();

        if (if_->expr_2)
            if_->expr_2->accept(this);
        Type *expr_2Type = this->getLastReturn();

        if (if_->expr_3)
            if_->expr_3->accept(this);
        Type *expr_3Type = this->getLastReturn();

        // Types Check
        nullptrCheck(expr_1Type, if_->expr_1->line_number, if_->expr_1->char_number);
        nullptrCheck(expr_2Type, if_->expr_2->line_number, if_->expr_2->char_number);
        nullptrCheck(expr_3Type, if_->expr_3->line_number, if_->expr_3->char_number);

        // Condition type Check
        if (compareTypes(expr_1Type, new TypeBool()) == false)
        {
            std::cout << "Error: wrong condition expression type: " << getNodeID(if_->expr_1) << std::endl
                      << "Expected: " << showner->show(new TypeBool()) << std::endl
                      << "But got: " << showner->show(expr_1Type) << std::endl
                      << "On the line: " << if_->line_number << std::endl;
            exit(1);
        }

        // Statements types Check
        if (compareTypes(expr_2Type, expr_3Type) == false && compareTypes(expr_3Type, expr_2Type) == false)
        {
            std::cout << "Error: different types of statements in the if statement: " << std::endl
                      << '"' << getNodeID(if_->expr_2) << "\" type is: " << showner->show(expr_2Type) << std::endl
                      << '"' << getNodeID(if_->expr_3) << "\" type is: " << showner->show(expr_3Type) << std::endl
                      << "On the line: " << if_->line_number << std::endl;
            exit(1);
        }

        if (compareTypes(expr_2Type, new TypeBottom()) == false)
        {
            expr_2Type = expr_3Type;
        }

        // Return if_ type
        this->setLastReturn(expr_2Type);
    }
```

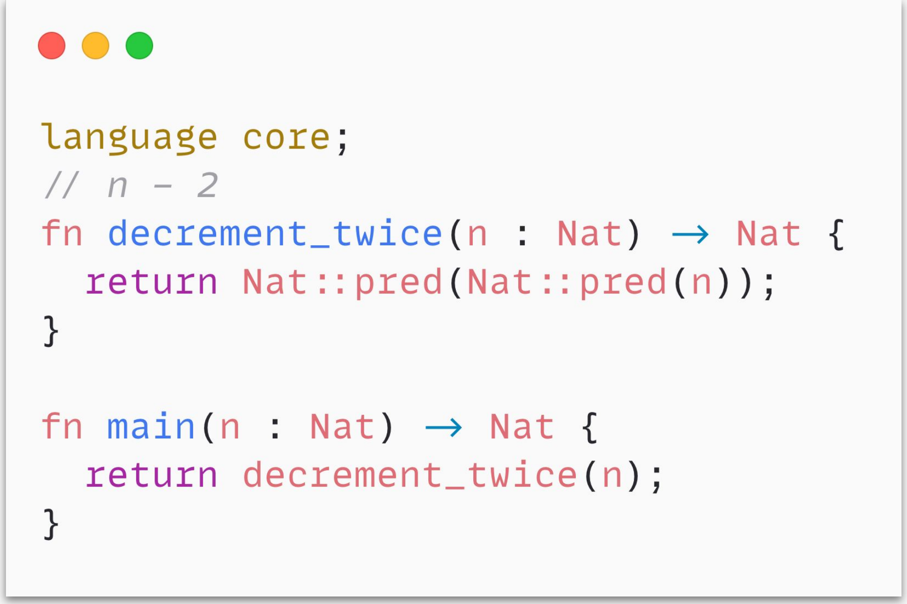


```
std::unordered_map<std::string,
 VariableDescriptions> decleardVariables;
std::vector<Type *> returnTypesList;
Type *lastReturn = nullptr;
Type *inType = nullptr;
PrintAbsyn *printer = new PrintAbsyn();
ShowAbsyn *showner = new ShowAbsyn();
```

Features of the language: successor and predecessor

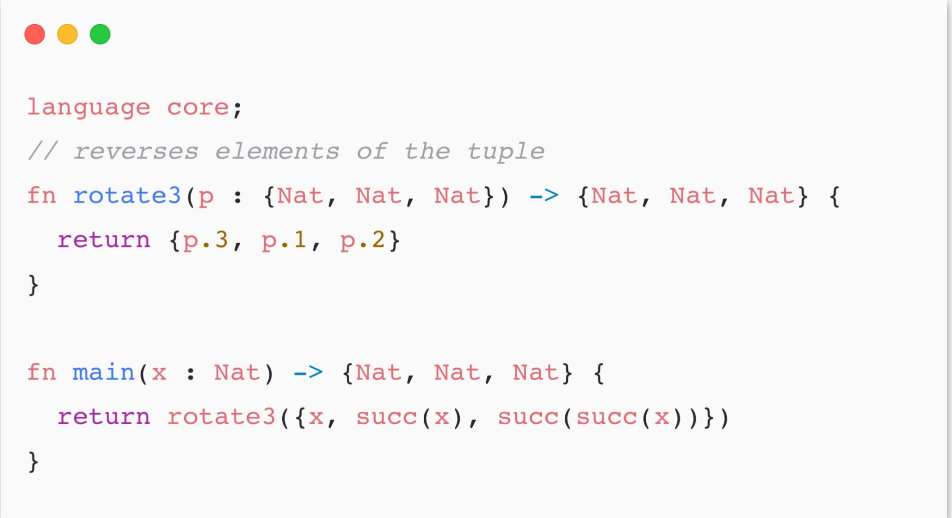


```
language core;
// n + 2
fn increment_twice(n : Nat) -> Nat {
  return succ(succ(n));
}

fn main(n : Nat) -> Nat {
  return increment_twice(n);
}
```

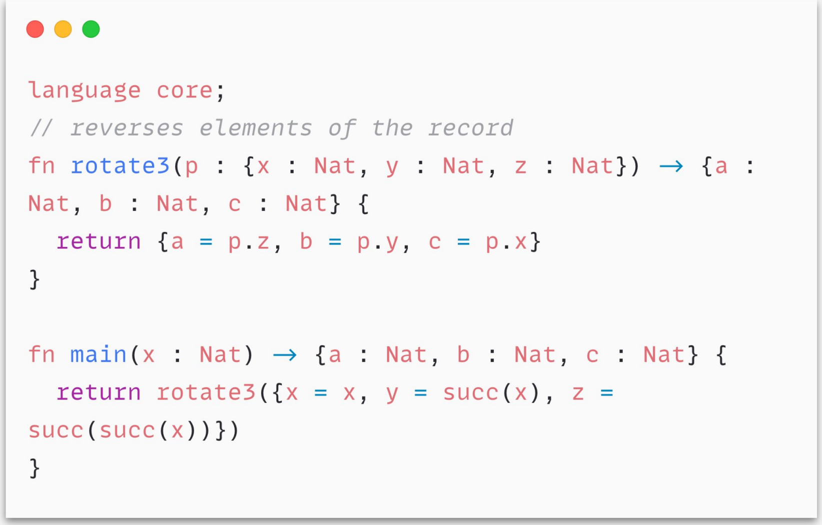


```
language core;
// n - 2
fn decrement_twice(n : Nat) → Nat {
  return Nat::pred(Nat::pred(n));
}

fn main(n : Nat) → Nat {
  return decrement_twice(n);
}
```

Features of the language: tuples and records

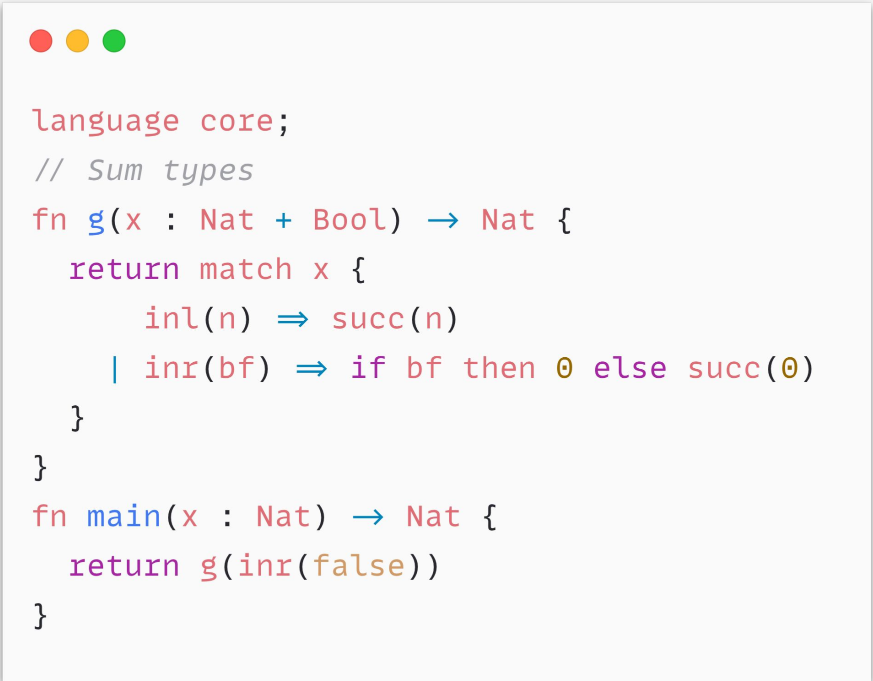


```
language core;
// reverses elements of the tuple
fn rotate3(p : {Nat, Nat, Nat}) -> {Nat, Nat, Nat} {
  return {p.3, p.1, p.2}
}

fn main(x : Nat) -> {Nat, Nat, Nat} {
  return rotate3({x, succ(x), succ(succ(x))})
}
```

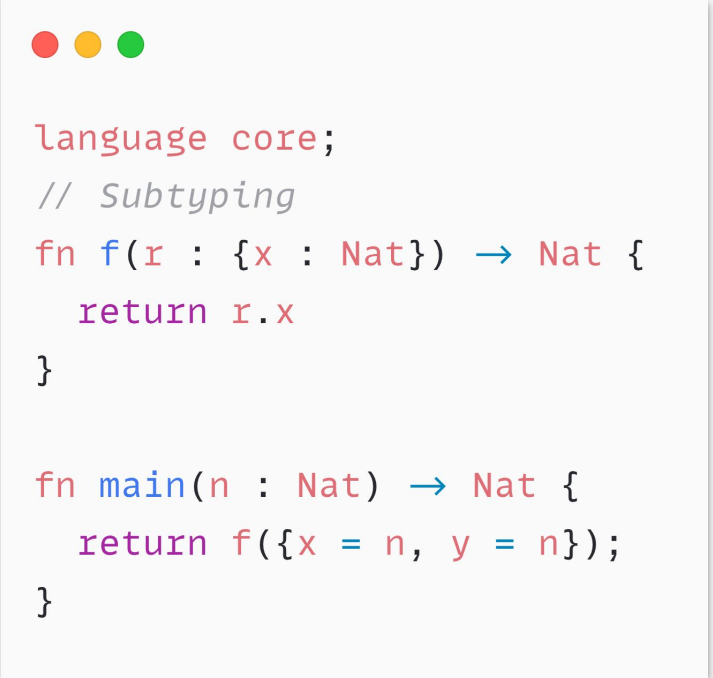


```
language core;
// reverses elements of the record
fn rotate3(p : {x : Nat, y : Nat, z : Nat}) → {a :
Nat, b : Nat, c : Nat} {
  return {a = p.z, b = p.y, c = p.x}
}

fn main(x : Nat) → {a : Nat, b : Nat, c : Nat} {
  return rotate3({x = x, y = succ(x), z =
succ(succ(x))})
}
```

Features of the language: sum of types and subtyping

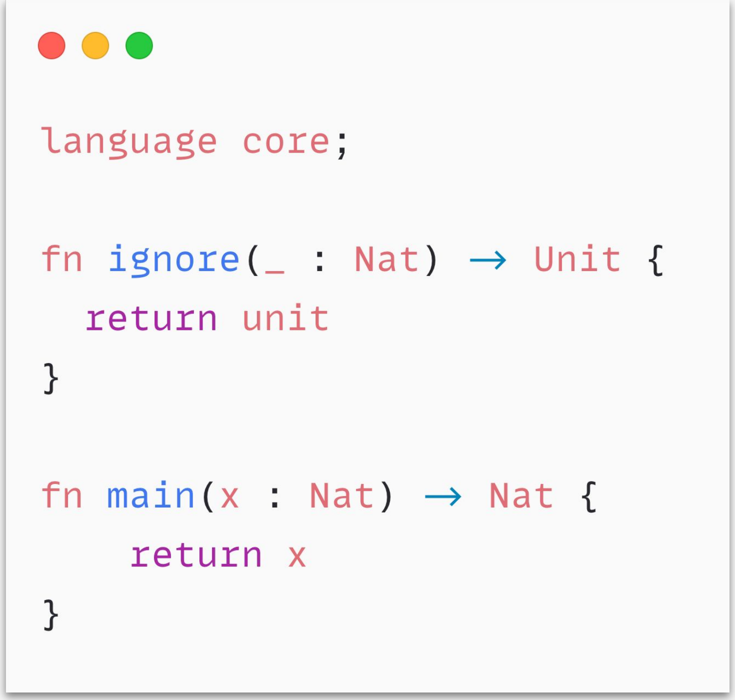


```
language core;
// Sum types
fn g(x : Nat + Bool) → Nat {
  return match x {
      inl(n) ⇒ succ(n)
    | inr(bf) ⇒ if bf then 0 else succ(0)
  }
}
fn main(x : Nat) → Nat {
  return g(inr(false))
}
```

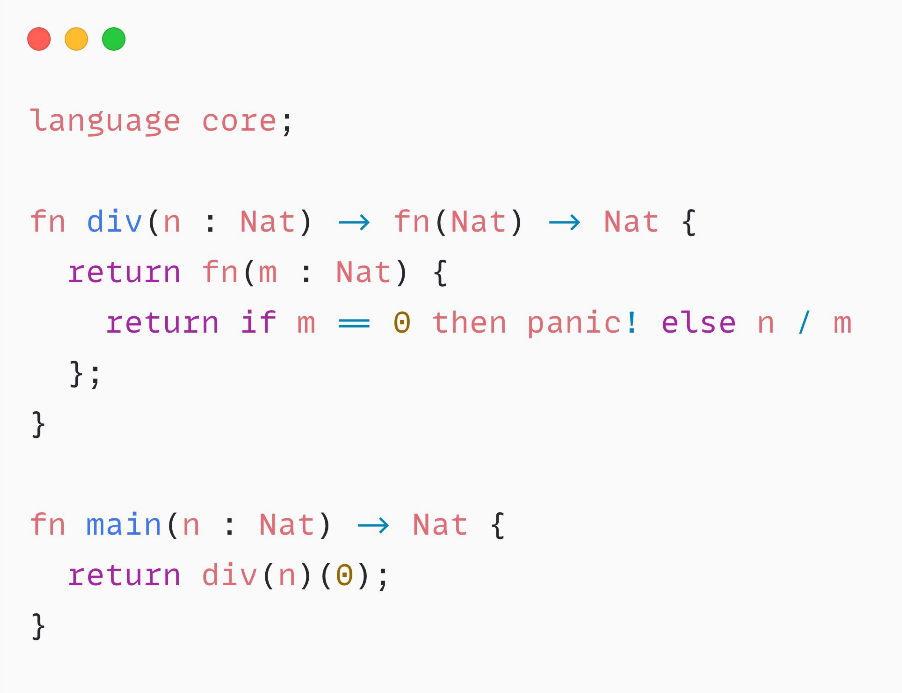


```
language core;
// Subtyping
fn f(r : {x : Nat}) → Nat {
  return r.x
}

fn main(n : Nat) → Nat {
  return f({x = n, y = n});
}
```

Features of the language: unit type, exception



```
language core;

fn ignore(_ : Nat) -> Unit {
  return unit
}

fn main(x : Nat) -> Nat {
    return x
}
```



```
language core;

fn div(n : Nat) -> fn(Nat) -> Nat {
  return fn(m : Nat) {
    return if m == 0 then panic! else n / m
  };
}

fn main(n : Nat) -> Nat {
  return div(n)(0);
}
```

Contributions

- Egor Shalagin – Static Analyzer, work with BNFC
- Mikhail Fedorov – Static Analyzer, testing
- Davlatkhodzha Magzumov – Static Analyzer, work with Makefiles
- *Saul Goodman – naming of the language*

Thank you for you attention