PURDUE UNIVERSITY

# Variables - Loops - Type Checking

Compilers: Principles And Practice

Tiark Rompf

# Where Were We?

What did we learn in the last class?

# Current Grammar

```
<op>    ::= [’*’ | ’/’ | ’+’ | ’-’ | ’<’ | ’>’ | ’=’ | ’!’]+
<bop>   ::= (’<’ | ’>’ | ’=’ | ’!’)[<op>]
<atom>  ::= <number>
          | ’(’<simp>’)’
          | <ident>
          | ’{’<exp>’}’
<uatom> ::= [<op>]<atom>
<cond>  ::= <simp><bop><simp>
<simp>  ::= <uatom>[<op><uatom>]*
          | ’if’ ’(’<cond>’)’ <simp> ’else’ <simp>
<exp>   ::= <simp>
          | ’val’ <ident> ’=’ <simp>’;’ <exp>
```

# Quiz

Are these valid syntaxes?

1. ```
   if (3 == 5) {
     2
   } * 4 else 8
   ```
   ```

# Quiz

Are these valid syntaxes?

1. ```
   if (3 == 5) {
     2
   } * 4 else 8
   ```

2. ```
   if (3 == 2)
     val x = 3; x
   else
     5
   ```

# Quiz

Are these valid syntaxes?

1. ```
   if (3 == 5) {
     2
   } * 4 else 8
   ```

2. ```
   if (3 == 2)
     val x = 3; x
   else
     5
   ```

Answer:

1. Yes
2. No: 'val' $x = 3; x$ is not a simple expression

# Let's Add Mutable Variables - Syntax

# Let's Add Mutable Variables - Syntax

```
<op>    ::= ['*' | '/' | '+' | '-' | '<' | '>' | '=' | '!']+
<bop>   ::= ('<' | '>' | '=' | '!')[<op>]
<atom>  ::= <number>
          | '('<simp>')'
          | <ident>
          | '{'<exp>'}'
<uatom> ::= [<op>]<atom>
<cond>  ::= <simp><bop><simp>
<simp>  ::= <uatom>[<op><uatom>]*
          | 'if' '('<cond>')' <simp> 'else' <simp>
          | <ident> '=' <simp>
<exp>   ::= <simp>
          | 'val' <ident> '=' <simp>';' <exp>
          | 'var' <ident> '=' <simp>';' <exp>
```

# Let's Add Mutable Variables - AST

# Let's Add Mutable Variables - AST

```
case class VarDec(name: String, value: Exp, body: Exp) extends Exp
case class VarAssign(name: String, value: Exp) extends Exp
```

# Let's Add Mutable Variables - Semantics

We can only assign to mutable variables, i.e. declared with 'var' (VarDec)

# Let's Add Mutable Variables - Semantics

We can only assign to mutable variables, i.e. declared with 'var' (VarDec)

```
type Value = Int

def eval(exp: Exp)(env: ValueEnv): Val = exp match {
  case VarDec(x, rhs, body) =>
    eval(body)(env.withVar(x, eval(rhs)(env)))
  case VarAssign(x, rhs) =>
    env.updateVar(x, eval(rhs)(env)) // return the new value
}
```

# Let's Add Loops - Syntax

# Let's Add Loops - Syntax

```
<op>    ::= ['*' | '/' | '+' | '-' | '<' | '>' | '=' | '!']+
<bop>   ::= ('<' | '>' | '=' | '!')[<op>]
<atom>  ::= <number>
          | '('<simp>')'
          | <ident>
          | '{'<exp>'}'
<uatom> ::= [<op>]<atom>
<cond>  ::= <simp><bop><simp>
<simp>  ::= <uatom>[<op><uatom>]*
          | 'if' '('<cond>')' <simp> 'else' <simp>
          | <ident> '=' <simp>
<exp>   ::= <simp>
          | 'val' <ident> '=' <simp>';' <exp>
          | 'var' <ident> '=' <simp>';' <exp>
          | 'while' '(' <cond> ')' <simp>; <exp>
```

# Let's Add Loops - AST

```
// Already defined
case class Cond(op: String, lop: Exp, rop: Exp) extends Exp
```

# Let's Add Loops - AST

```scala
// Already defined
case class Cond(op: String, lop: Exp, rop: Exp) extends Exp

case class While(cond: Cond, lbody: Exp, body: Exp) extends Exp
```

# Let's Add Loops - Semantics

```scala
type Value = Int

def eval(exp: Exp)(env: ValueEnv): Val = exp match {
  case While(Cond(op, l, r), lbody, body) =>
    while (evalCond(op)(eval(l)(env), eval(r)(env))) {
      eval(lbody)(env)
    }
    eval(body)(env)
}
```

# x86 Flags And Jump - Compile Loops

```
trans(While(Cond(op, l, r), lbody, body), 0)(Map())
```

# x86 Flags And Jump - Compile Loops

```
trans(While(Cond(op, l, r), lbody, body), 0)(Map())
```

In order to compile while statement, we are going to follow this idea:

```
  jmp loop_cond
loop_body:
  ... # code for lbody
loop_cond:
  ... # code for l and r
  cmpq <r>, <l>
  j<op> loop_body # the jump operation depends on 'op'
  ... # code for body
```

# x86 Flags And Jump - Compile Loops

```
trans(While(Cond(op, l, r), lbody, body), 0)(Map())
```

In order to compile while statement, we are going to follow this idea:

```
  jmp loop_cond
loop_body:
  ... # code for lbody
loop_cond:
  ... # code for l and r
  cmpq <r>, <l>
  j<op> loop_body # the jump operation depends on 'op'
  ... # code for body
```

How would we compile a do-while loop?

# x86 Flags And Jump - Compile Loops

```
trans(While(Cond(op, l, r), lbody, body), 0)(Map())
```

In order to compile while statement, we are going to follow this idea:

```
  jmp loop_cond
loop_body:
  ... # code for lbody
loop_cond:
  ... # code for l and r
  cmpq <r>, <l>
  j<op> loop_body # the jump operation depends on 'op'
  ... # code for body
```

How would we compile a do-while loop?

Answer: omit the unconditional jump

# We Can Write, Parse, and Compile Nice Code!!

```
var x = 2;
var y = 0;
while (y < 5) {
  x = x * x;
  y = y + 1
};
x
```

# We Can Write, Parse, and Compile Nice Code!!

```
var x = 2;
var y = 0;
while (y < 5) {
  x = x * x;
  y = y + 1
};
x
```

Can we really???

# We Can Write, Parse, and Compile Nice Code!!

```
var x = 2;
var y = 0;
while (y < 5) {
  x = x * x;
  y = y + 1
};
x
```

Can we really???

We need to modify our program slightly.

# We Can Write, Parse, and Compile Nice-ish Code!!

```
var x = 2;
var y = 0;
while (y < 5) {
  val dummy = x = x * x;
  y = y + 1
};
x
```

# We Can Write, Parse, and Compile Nice-ish Code!!

```
var x = 2;
var y = 0;
while (y < 5) {
  val dummy = x = x * x;
  y = y + 1
};
x
```

Let's modify our grammar slightly instead!!

# Grammar - Syntactic Sugar

```
<op>    ::= ['*' | '/' | '+' | '-' | '<' | '>' | '=' | '!']+
<bop>   ::= ('<' | '>' | '=' | '!')[<op>]
<atom>  ::= <number>
          | '('<simp>')'
          | <ident>
          | '{'<exp>'}'
<uatom> ::= [<op>]<atom>
<cond>  ::= <simp><bop><simp>
<simp>  ::= <uatom>[<op><uatom>]*
          | 'if' '('<cond>')' <simp> ['else' <simp>]
          | <ident> '=' <simp>
<exp>   ::= <simp>[';'<exp>]
          | 'val' <ident> '=' <simp>';'<exp>
          | 'var' <ident> '=' <simp>';'<exp>
          | 'while' '(' <cond> ')' <simp>';'<exp>
```

# Grammar - Syntactic Sugar

Syntax sugar constructs are constructs that can be syntactically translated to other existing constructs. Syntactic sugar does not offer additional expressive power to the programmer; only some syntactic convenience

# Grammar - Syntactic Sugar

Syntax sugar constructs are constructs that can be syntactically translated to other existing constructs. Syntactic sugar does not offer additional expressive power to the programmer; only some syntactic convenience

```
x = x + 1;
y = y + 1
```

# Grammar - Syntactic Sugar

Syntax sugar constructs are constructs that can be syntactically translated to other existing constructs. Syntactic sugar does not offer additional expressive power to the programmer; only some syntactic convenience

```
x = x + 1;
y = y + 1
```

rather than

```
val dummy = x = x + 1;
y = y + 1
```

# Unit Type

```
val tmp = if (x > 0)
  x = x - 1
else
  0; // Won't be used
val y = x * 5;
y
```

## Unit Type

```
val tmp = if (x > 0)
  x = x - 1
else
  0; // Won't be used
val y = x * 5;
y
```

is now transformed into

```
if (x > 0)
  x = x - 1;
val y = x * 5;
x
```

# Unit Type

```
val tmp = if (x > 0)
  x = x - 1
else
  0; // Won't be used
val y = x * 5;
y
```

is now transformed into

```
if (x > 0)
  x = x - 1;
val y = x * 5;
x
```

What is the type of this if expression?

# AST Of Sugared Expressions

```
x = x + 1;
y = y + 1
```

# AST Of Sugared Expressions

```
x = x + 1;
y = y + 1
```

```
Let("tmp$1",
   VarAssign("x", Prim("+", Ref("x"), Lit(1)))
      VarAssign("y", Prim("+", Ref("y"), Lit(1)))
)
```

# AST Of Sugared Expressions

```
if (x > 0)
  x = x - 1;
val y = x * 5;
x
```

# AST Of Sugared Expressions

```
if (x > 0)
  x = x - 1;
val y = x * 5;
x
```

```
Let("tmp$1",
  If(Cond(">", Ref("x"), Lit(0)),
    VarAssign("x", Prim("-", Ref("x"), Lit(1))),
    Lit(())),
  Let("y", Prim("*", Ref("x"), Lit(5)), Ref("x"))
)
```

# About Types

What is a type?

# About Types

What is a type?

- A set of values: e.g. true, false

## About Types

What is a type?

- A set of values: e.g. true, false
- A set of operations on those values: e.g. !, &&, ...

# About Types

What is a type?

- A set of values: e.g. true, false
- A set of operations on those values: e.g. !, &&, ...

# About Types

What is a type?

- A set of values: e.g. true, false
- A set of operations on those values: e.g. !, &&, ...

Why do we need types?

- Help to structure and understand a program

# About Types

What is a type?

- A set of values: e.g. true, false
- A set of operations on those values: e.g. !, &&, ...

Why do we need types?

- Help to structure and understand a program
- Can prevent some kinds of errors

## Our Grammar, Typed

```
<op>     ::= [’*’ | ’/’ | ’+’ | ’-’ | ’<’ | ’>’ | ’=’ | ’!’]+
<type>   ::= <ident>
<bool>   ::= ’true’ | ’false’
<atom>   ::= <number> | <bool> | ’()’
           | ’(’<simp>’)’
           | <ident>
           | ’{’<exp>’}’
<uatom> ::= [<op>]<atom>
<simp>   ::= <uatom>[<op><uatom>]*
           | ’if’ ’(’<simp>’)’ <simp> [’else’ <simp>]
           | <ident> ’=’ <simp>
<exp>    ::= <simp>[’;’<exp>]
           | ’val’ <ident> [’:’ <type>] ’=’ <simp>’;’<exp>
           | ’var’ <ident> [’:’ <type>] ’=’ <simp>’;’<exp>
           | ’while’ ’(’ <simp> ’)’ <simp>’;’<exp>
```

# Our AST, Typed

First, we modify our AST to handle the new grammar:

```
abstract class Type
// Definition later

case class Lit(x: Any) extends Exp
case class Let(name: Int, tp: Type, v: Exp, b: Exp) extends Exp
case class VarDec(name: Int, tp: Type, v: Exp, b: Exp) extends Exp
case class If(cond: Exp, tBranch: Exp, eBranch: Exp) extends Exp
case class While(cond: Exp, lbody: Exp, body: Exp) extends Exp
```

# Inference Rules

```
Env |- e: T
```

Means that in the environment ‘Env’, the expression ‘e’ is of type ‘T’

# Inference Rules

```
Env |- e: T
```

Means that in the environment ‘Env’, the expression ‘e’ is of type ‘T’

Env is the typing environment: it is a mapping between identifier and type.

Env,id:T is the environment that contains all information in ‘Env’ plus the mapping from ‘id’ to ‘T’

Note: Env is often represented with $\Gamma$.

# Inference Rules

```
Env |- e: T
```

Means that in the environment ‘Env’, the expression ‘e’ is of type ‘T’

Env is the typing environment: it is a mapping between identifier and type.

Env,id:T is the environment that contains all information in ‘Env’ plus the mapping from ‘id’ to ‘T’

Note: Env is often represented with Γ.

```
 conditions
------------ [Name of the rule]
 conclusion
```

# Type Checking

The type checking step will be part of the semantic analyzer.

The key point to understand is that types represent an abstract value, and inference rules are the set of operations on these values.

Therefore, the implementation is going to be very similar to eval or analyze.

# Where Are We?

We added variables, loops and some syntax sugar to our language.

We introduced types and typing rules.

We saw types as abstract values which can be computed. We also defined a simplified type checking algorithm.

# Where Are We?

We added variables, loops and some syntax sugar to our language.

We introduced types and typing rules.

We saw types as abstract values which can be computed. We also defined a simplified type checking algorithm.

Questions?