CS107: Variables, Loops, and Type Checking

Guannan Wei
`guannan.wei@tufts.edu`
Jan 27, 2026

Spring 2026

Tufts University

Recap

What did we learn last time?

Recap

What did we learn last time?

- Syntax error vs semantic error
- Better error reporting
- Interpreting and compiling conditionals

Current Grammar

The grammar so far:

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

Quiz

Is this valid syntax?

1)

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

Quiz

Is this valid syntax?

1)

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

2)

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

Quiz

Is this valid syntax?

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

Missing Features

- What are still missing from our language?

Missing Features

- What are still missing from our language?
- Let's add mutable variables!

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

Example

Example:

```
var x = 2;
x = x * x
```

Let’s Add Mutable Variables - AST

New AST nodes:

```
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
  // previous cases omitted
  case VarDec(x, rhs, body) =>
    val v = eval(rhs)(env)
    eval(body)(env.withVar(x, v))
  case VarAssign(x, rhs) =>
    val v = eval(rhs)(env)
    env.updateVar(x, v)
```

- What would be the value of assignment?
    - Unit or the assigned value

Let's Add Loops - Syntax

- Let's then add loops!

Let's Add Loops - Syntax

- Let's then add loops!

```
<op>    ::= ['+' | '-' | '*' | '/']+
<bop>   ::= '==' | '!=' | '<' | '>' | '<=' | '>='
<atom>  ::= <number>
          | <ident>
          | '('<simp>')'
          | '{'<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> // new
```

Let's Add Loops - AST

```scala
// 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

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

Let's Add Loops - Semantics

- Implementing `while` in the interpreter using Scala's `while`:

```
type Value = Int

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

- Note that the `ValueEnv` is mutable, so changes in the loop body persist.

x86 Flags And Jump

Recap: how to compile conditionals?

```
trans(If(Cond("==", 1, 0), 2, 3), 0)(Map())
```

```
# begin code generated
  movq $1, %rbx # generate code that compute l, stored in %rbx
  movq $0, %rcx # generate code that compute r, stored in %rcx
  cmpq %rcx, %rbx
  je if1_then
  movq $3, %rbx # generate code for eBranch, store result in %rbx
  jmp if1_end
if1_then:
  movq $2, %rbx # generate code for tBranch, store result in %rbx
if1_end:        # end code generated
  movq %rbx, %rax
  ret
```

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?

```
  <atom>  ::= <number> | <ident> | '('<simp>')' | '{'<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>
```

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

What has to been written is actually:

```
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!!

What has to been written is actually:

```
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>   ::= '==' | '!=' | '<' | '>' | '<=' | '>='
<atom>  ::= <number>
          | <ident>
          | '('<simp>')'
          | '{'<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>
```

What have been changed?

Grammar - Syntactic Sugar

- Syntax sugar constructs are constructs that can be syntactically translated to other existing core 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 core 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 core 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
```

now can be written as

```
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
```

now can be written as

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

What is the type of this if expression?

Unit Type

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

now can be written as

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

What is the type of this if expression?

We cannot always meaningfully synthesize a value for the else-branch. So we introduce a unit type and its sole value `()`.

AST of Sugared Expressions

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

AST of Sugared Expressions

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

Parser produces:

```
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
```

Parser produces:

```
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")))
```

AST of Sugared Expressions

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

Parser produces:

```
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")))
```

You will implement parsing with syntactic sugars in Project 3.

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

What is a type?

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

What is a type?

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

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

What is a type?

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

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

What is a type?

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

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

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 structure and understand a program

Introducing Type Systems

- So far we talked about syntax and dynamic semantics (specified as interpretation or translation).
- Languages typically have a static semantics, often specified as a type system.

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 structure and understand a program
- Can prevent some kinds of errors or undefined behaviors

Our Grammar, Typed

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

Example

```
var x: Int = 2;
val y: Int = 0;
x = y
```

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: String, tp: Type, v: Exp, b: Exp) extends Exp
case class VarDec(name: String, 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

Typing judgments: we write

$$\Gamma \vdash e : T$$

to assert that in the environment Γ, the expression e is of type T.

Inference Rules

Typing judgments: we write

$$\Gamma \vdash e : T$$

to assert that in the environment Γ, the expression e is of type T.

- Γ is the **typing environment**: It stores knowledge about identifiers available at compile time, as a finite mapping from identifiers to types. Grammar:

  ```
  Γ ::= ∅ | Γ, id:T
  ```

- We write $\emptyset$ for the empty typing environment, and
- $\Gamma, id : T$ to extend the typing environment Γ with a new mapping from id to type T.

Inference Rules

- A type system consists of a set of inductively defined **inference rules**.
- These rules define how to form an instance of typing judgments, i.e. proving that an expression has a certain type in a certain environment.
- General form of inference rules:

$$\frac{\mathit{condition1} \qquad \mathit{condition2} \qquad \ldots}{\mathit{conclusion}}\ \textsc{name of the rule}$$

Type Checking

The type checking realizes typing rules as 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` .

Inference Rules

1) Lit: i is an Int, b is a Boolean

$$\Gamma \vdash \texttt{Lit}(i) : \texttt{Int} \ \textsc{int} \qquad\qquad \Gamma \vdash \texttt{Lit}(b) : \texttt{Boolean} \ \textsc{boolean}$$

$$\Gamma \vdash \texttt{Lit}(\texttt{()}) : \texttt{Unit} \ \textsc{Unit}$$

We call inference rules without conditions **axioms**.

Inference Rules

1) Lit: i is an Int, b is a Boolean

$$\Gamma \vdash \texttt{Lit}(i) : \texttt{Int} \ \textsc{int} \qquad \Gamma \vdash \texttt{Lit}(b) : \texttt{Boolean} \ \textsc{boolean}$$

$$\Gamma \vdash \texttt{Lit}(\texttt{()}) : \texttt{Unit} \ \textsc{Unit}$$

We call inference rules without conditions **axioms**.

2) Unary: $op \in \{\texttt{"+"}, \texttt{"-"}\}$

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{Unary}(op, e) : \texttt{Int}} \ \textsc{intunop}$$

3) Prim:

- $op \in \{\texttt{"+"},\ \texttt{"-"},\ \texttt{"*"},\ \texttt{"/"}\}$
- $bop \in \{\texttt{"="},\ \texttt{"}\neq\texttt{"},\ \texttt{"}\leqslant\texttt{"},\ \texttt{"}\geqslant\texttt{"},\ \texttt{"<"},\ \texttt{">"}\}$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash \texttt{Prim}(op, e_1, e_2) : \texttt{Int}}\ \textsc{IntOp} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash \texttt{Prim}(bop, e_1, e_2) : \texttt{Boolean}}\ \textsc{BoolOp}$$

Inference Rules

4) Immutable variables

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{Let}(x, T_1, e_1, e_2) : T_2}\ \textsc{Let} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash \texttt{Ref}(x) : T}\ \textsc{Ref}$$

Where Are We?

- We added variables, loops and some syntactic 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 syntactic 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?