

# COMP-520 – Review lecture

Vincent Foley-Bourgon

Sable Lab  
McGill University

Winter 2017

# Plan

- ▶ We'll go over the different concepts we saw in class

# Plan

- ▶ We'll go over the different concepts we saw in class
- ▶ **You** will have to provide the answers

# Plan

- ▶ We'll go over the different concepts we saw in class
- ▶ **You** will have to provide the answers
- ▶ I know the names of many of you; if you don't want to be called out, volunteer an answer :)

# Compiler overview

What is a compiler?

# What is a compiler?

An *automated* program that *translates* programs written in a *source language* into *equivalent* programs in a *target language*.

# Compiler vs interpreter?



# Compiler vs interpreter?

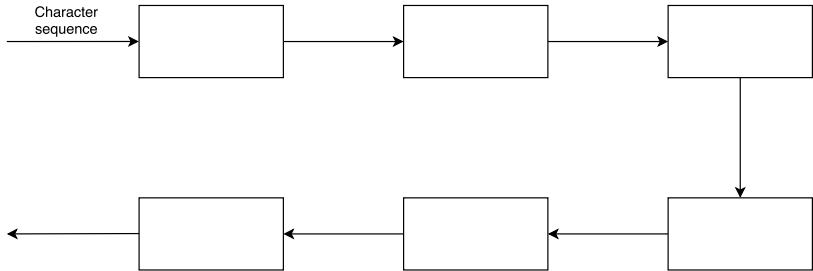
- ▶ Compiler: *translate* a program (execute the result later)
- ▶ Interpreter: *execute* a program immediately

# AOT vs JIT?

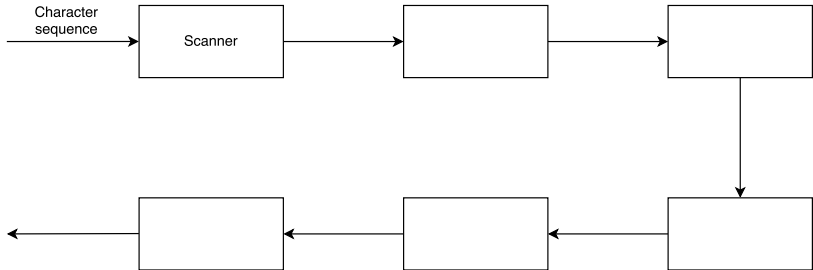
# AOT vs JIT?

- ▶ AOT: compile everything now, execute later
- ▶ JIT: execute now (interpreter), compile the hot parts during execution

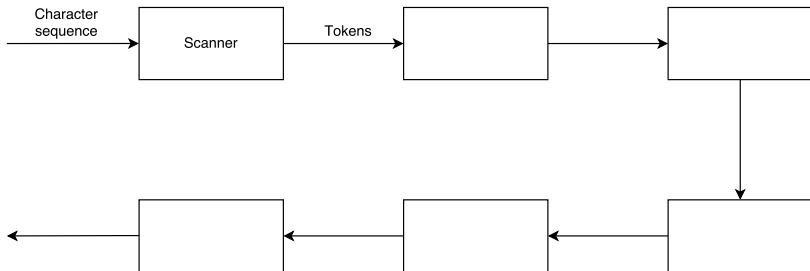
# Phases of the compilers



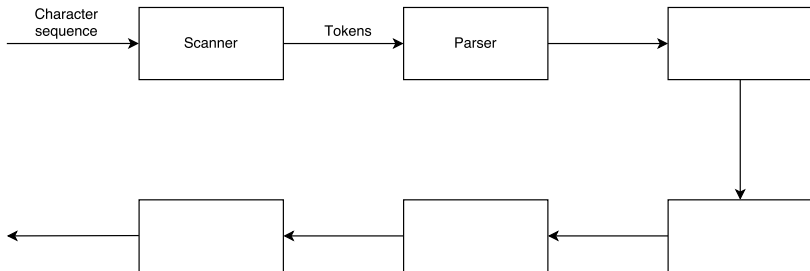
# Phases of the compilers



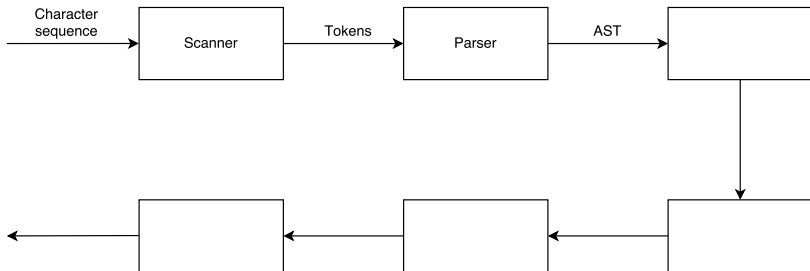
# Phases of the compilers



# Phases of the compilers

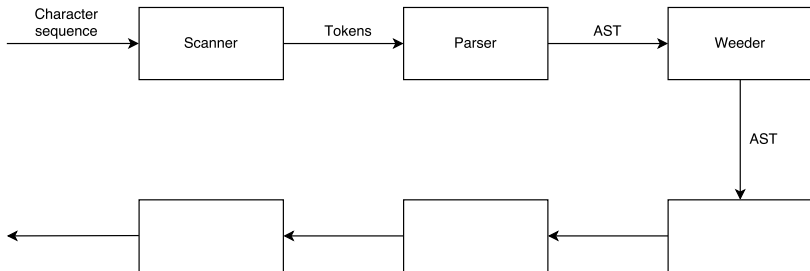


# Phases of the compilers

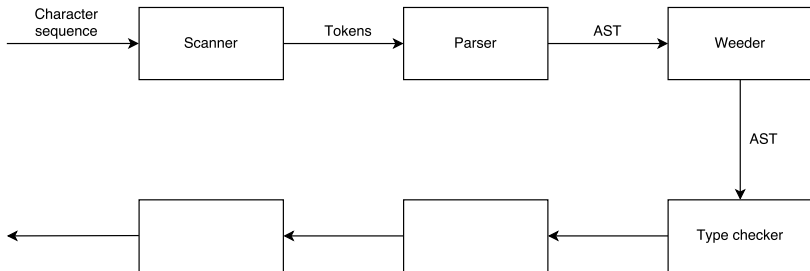




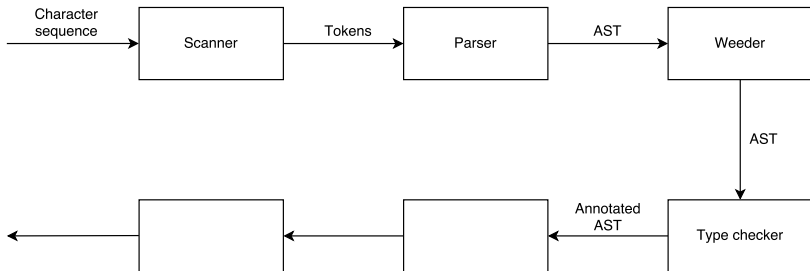
# Phases of the compilers



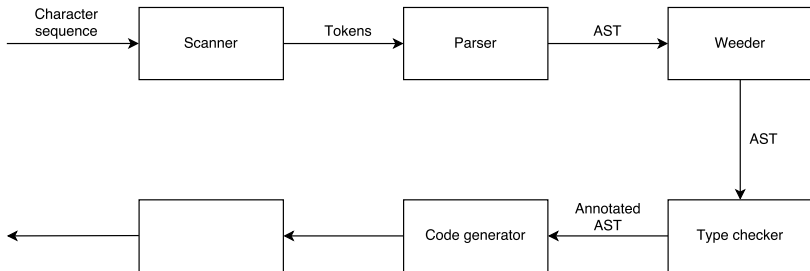
# Phases of the compilers



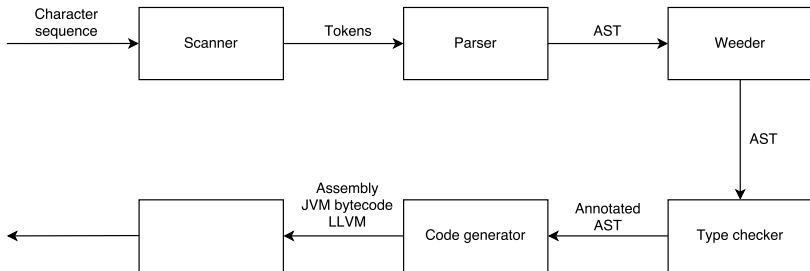
# Phases of the compilers



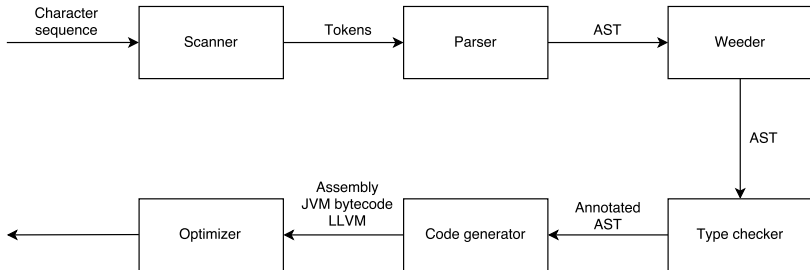
# Phases of the compilers



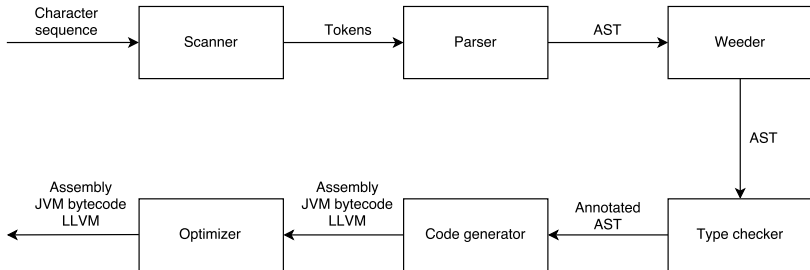
# Phases of the compilers



# Phases of the compilers



# Phases of the compilers



Scanner



# Scanner generalities

- ▶ What is the input of a scanner?

# Scanner generalities

- ▶ What is the input of a scanner? **Characters**

# Scanner generalities

- ▶ What is the input of a scanner? **Characters**
- ▶ What is the output of a scanner?

# Scanner generalities

- ▶ What is the input of a scanner? **Characters**
- ▶ What is the output of a scanner? **Tokens**

# Scanner generalities

- ▶ What is the input of a scanner? **Characters**
- ▶ What is the output of a scanner? **Tokens**
- ▶ What formalism did we use to specify scanners?

# Scanner generalities

- ▶ What is the input of a scanner? **Characters**
- ▶ What is the output of a scanner? **Tokens**
- ▶ What formalism did we use to specify scanners? **Regular expressions**

# Regular expressions

What are the 5 building blocks of regular expressions?

- ▶ C
- ▶ E
- ▶ C
- ▶ A
- ▶ R

# Regular expressions

What are the 5 building blocks of regular expressions?

- ▶ Character 'c'
- ▶ E
- ▶ C
- ▶ A
- ▶ R



# Regular expressions

What are the 5 building blocks of regular expressions?

- ▶ Character 'c'
- ▶ Empty string  $\epsilon$
- ▶ C
- ▶ A
- ▶ R

# Regular expressions

What are the 5 building blocks of regular expressions?

- ▶ Character 'c'
- ▶ Empty string  $\epsilon$
- ▶ Concatenation **AB**
- ▶ A
- ▶ R

# Regular expressions

What are the 5 building blocks of regular expressions?

- ▶ Character 'c'
- ▶ Empty string  $\epsilon$
- ▶ Concatenation **AB**
- ▶ Alternation **A|B**
- ▶ R

# Regular expressions

What are the 5 building blocks of regular expressions?

- ▶ Character ' $c$ '
- ▶ Empty string  $\epsilon$
- ▶ Concatenation  $\mathbf{AB}$
- ▶ Alternation  $\mathbf{A|B}$
- ▶ Repetition  $\mathbf{A^*}$

# Regular expressions

More regular expressions

- ▶ Optional

# Regular expressions

More regular expressions

- ▶ Optional  $A? = A \mid \epsilon$

# Regular expressions

More regular expressions

- ▶ Optional  $A? = A \mid \epsilon$
- ▶ One-or-more

# Regular expressions

More regular expressions

- ▶ Optional  $A? = A \mid \epsilon$
- ▶ One-or-more  $A+ = A(A^*)$



# Regular expressions

More regular expressions

- ▶ Optional  $A? = A | \epsilon$
- ▶ One-or-more  $A+ = A(A^*)$
- ▶ Range of characters

# Regular expressions

More regular expressions

- ▶ Optional  $A? = A | \epsilon$
- ▶ One-or-more  $A+ = A(A^*)$
- ▶ Range of characters  $[a-c] = 'a' | 'b' | 'c'$

# Scanner

How does flex match tokens?

# Scanner

How does flex match tokens?

**TRY ALL THE REGEXES!!1**



# Scanner

How does flex handle multiple matches?

# Scanner

How does flex handle multiple matches?

- ▶ Longest match rule (e.g. var vs variance)

# Scanner

How does flex handle multiple matches?

- ▶ Longest match rule (e.g. var vs variance)
- ▶ First match rule (e.g. keywords vs identifiers)

# Scanner

How does flex make regular expressions executable?



# Scanner

How does flex make regular expressions executable?

Regular expression  $\rightarrow$  NFA  $\rightarrow$  DFA

# Regular languages

Given a language, what is one sign that it is not a regular language?

# Regular languages

Given a language, what is one sign that it is not a regular language?

Arbitrary nesting (e.g. parentheses, control structures)

Regular languages cannot be defined recursively.

# Parser

# Parser generalities

- ▶ What is the input of a parser?

# Parser generalities

- ▶ What is the input of a parser? **Tokens**

# Parser generalities

- ▶ What is the input of a parser? **Tokens**
- ▶ What is the output of a parser?

# Parser generalities

- ▶ What is the input of a parser? **Tokens**
- ▶ What is the output of a parser? **Syntax tree (abstract or concrete)**



# Parser generalities

- ▶ What is the input of a parser? **Tokens**
- ▶ What is the output of a parser? **Syntax tree (abstract or concrete)**
- ▶ What formalism did we use to specify parsers?

# Parser generalities

- ▶ What is the input of a parser? **Tokens**
- ▶ What is the output of a parser? **Syntax tree (abstract or concrete)**
- ▶ What formalism did we use to specify parsers?  
**Context-free grammars**

# Context-free grammars

What are the 4 components of context-free grammars?

- ▶ T
- ▶ N
- ▶ P
- ▶ S

# Context-free grammars

What are the 4 building blocks of context-free grammars?

- ▶ Terminals (tokens)
- ▶ N
- ▶ P
- ▶ S

# Context-free grammars

What are the 4 building blocks of context-free grammars?

- ▶ Terminals (tokens)
- ▶ Non-terminals (e.g. *stmt* or *expr*)
- ▶ P
- ▶ S

# Context-free grammars

What are the 4 building blocks of context-free grammars?

- ▶ Terminals (tokens)
- ▶ Non-terminals (e.g. *stmt* or *expr*)
- ▶ Productions (e.g. *stmt*  $\rightarrow$  *PRINT* '(' *expr* ')')
- ▶ S

# Context-free grammars

What are the 4 building blocks of context-free grammars?

- ▶ Terminals (tokens)
- ▶ Non-terminals (e.g. *stmt* or *expr*)
- ▶ Productions (e.g. *stmt*  $\rightarrow$  *PRINT* '(' *expr* ')')
- ▶ Start symbol

# Context-free grammars

When is a grammar ambiguous?



# Context-free grammars

When is a grammar ambiguous?

**When *at least one sentence that has more than one derivation.***

# Ambiguous grammar

Grammar:  $E \rightarrow id \mid E '+' E$

Program:  $id + id + id$

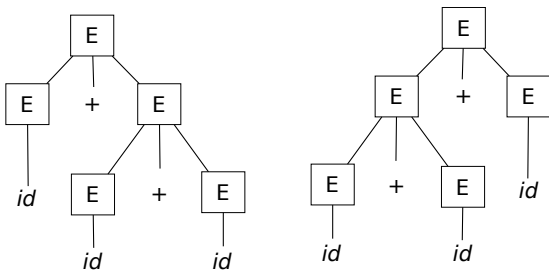
What are the two derivations for this sentence?

# Ambiguous grammar

Grammar:  $E \rightarrow id \mid E '+' E$

Program:  $id + id + id$

What are the two derivations for this sentence?



# Ambiguous grammar

What are the two ways to fix this ambiguity?

# Ambiguous grammar

What are the two ways to fix this ambiguity?

Factoring the grammar:

$$E = E \text{ '+' } T \mid T;$$
$$T = \text{id};$$

# Ambiguous grammar

What are the two ways to fix this ambiguity?

Factoring the grammar:

```
E = E '+' T | T;
```

```
T = id;
```

Precedence+associativity declarations:

```
%left '+'
```

```
E = id | E '+' E;
```

# Parsers

What do LL(1) and LR(1) mean?

# Parsers

What do LL(1) and LR(1) mean?

- ▶ LL(1): left-to-right processing, **left-most derivation**, one token of lookahead
- ▶ LR(1): left-to-right processing, **right-most derivation**, one token of lookahead



# Parsers

What is a left-most derivation? A right-most derivation?

# Parsers

What is a left-most derivation? A right-most derivation?

```
stmt = IF expr THEN stmt ENDIF  
      | PRINT expr  
expr = ID
```

# Parsers

What is a left-most derivation? A right-most derivation?

```
stmt = IF expr THEN stmt ENDIF
      | PRINT expr
expr = ID
```

```
if x then print x endif
```

# Parsers

What is a left-most derivation? A right-most derivation?

```
stmt = IF expr THEN stmt ENDIF
      | PRINT expr
expr = ID
```

```
if x then print x endif
```

```
// left-most derivation
IF expr THEN stmt ENDIF ==>
```

# Parsers

What is a left-most derivation? A right-most derivation?

```
stmt = IF expr THEN stmt ENDIF
      | PRINT expr
expr = ID
```

```
if x then print x endif
```

```
// left-most derivation
IF expr THEN stmt ENDIF ==>
  IF ID THEN stmt ENDIF
```

# Parsers

What is a left-most derivation? A right-most derivation?

```
stmt = IF expr THEN stmt ENDIF
      | PRINT expr
expr = ID
```

```
if x then print x endif
```

```
// left-most derivation
IF expr THEN stmt ENDIF ==>
  IF ID THEN stmt ENDIF
```

```
// right-most derivation
IF expr THEN stmt ENDIF ==>
```

# Parsers

What is a left-most derivation? A right-most derivation?

```
stmt = IF expr THEN stmt ENDIF
      | PRINT expr
expr = ID
```

```
if x then print x endif
```

```
// left-most derivation
IF expr THEN stmt ENDIF ==>
  IF ID THEN stmt ENDIF
```

```
// right-most derivation
IF expr THEN stmt ENDIF ==>
  IF expr THEN PRINT expr ENDIF
```

# Parsers

What are the two types of parser we saw in class?

- ▶ T
- ▶ B



# Parsers

What are the two types of parser we saw in class?

- ▶ Top-down
- ▶ B

# Parsers

What are the two types of parser we saw in class?

- ▶ Top-down
- ▶ Bottom-up

# Parsers

What is the difference between top-down and bottom-up?

# Parsers

What is the difference between top-down and bottom-up?

- ▶ Top-down: start symbol  $\downarrow$  leaves
- ▶ Bottom-up: leaves  $\uparrow$  start symbol

# Recursive descent parser

```
// Grammar  
stmt = ID '=' expr ';' |  
      | PRINT expr ';' |  
      | ...
```

# Recursive descent parser

```
// Grammar
stmt = ID '=' expr ';'
      | PRINT expr ';'
      | ...

// Python code
def stmt():
    next_tok = peek()
    if next_tok == TOK_ID:
        id = consume(TOK_ID)
        consume(TOK_EQ)
        e = expr()
        consume(TOK_SEMI)
        return astnode(AST_ASSIGN, lhs=id, rhs=e)
    elif next_tok == TOK_PRINT:
        consume(TOK_PRINT)
        e = expr()
        consume(TOK_SEMI)
        return astnode(AST_PRINT, expr=e)
    elif ...
```

# Bottom-up parsers

What are the three actions of a bottom-up parser?

- ▶ S
- ▶ R
- ▶ A

# Bottom-up parsers

What are the three actions of a bottom-up parser?

- ▶ Shift (move a token from input to stack)
- ▶ R
- ▶ A



# Bottom-up parsers

What are the three actions of a bottom-up parser?

- ▶ Shift (move a token from input to stack)
- ▶ Reduce (replace the rhs of a production that's on top of the stack with its lhs)
- ▶ A

# Bottom-up parsers

What are the three actions of a bottom-up parser?

- ▶ Shift (move a token from input to stack)
- ▶ Reduce (replace the rhs of a production that's on top of the stack with its lhs)
- ▶ Accept

# Bottom-up parsers

What type of conflict is exhibited in this grammar?

```
%{
```

```
%}
```

```
%token ID
```

```
%start start
```

```
%%
```

```
start: rule1 | rule2
```

```
rule1: ID
```

```
rule2: ID
```

```
%%
```

# Bottom-up parsers

What type of conflict is exhibited in this grammar?

```
%{  
%}  
  
%token ID  
%start start  
  
%%  
start: rule1 | rule2  
rule1: ID  
rule2: ID  
%%
```

**Reduce/reduce**

# Bottom-up parsers

What type of conflict is exhibited in this grammar?

```
%{  
%}  
  
%token ID  
%start start  
  
%%  
start: ID ID | rule1 ID  
rule1: ID  
%%
```

# Bottom-up parsers

What type of conflict is exhibited in this grammar?

```
%{  
%}  
  
%token ID  
%start start  
  
%%  
start: ID ID | rule1 ID  
rule1: ID  
%%
```

**Shift/reduce**

AST

# Concrete syntax tree

- ▶ What is a CST?



# Concrete syntax tree

- ▶ What is a CST? **The tree that traces a parser derivation**

# Concrete syntax tree

- ▶ What is a CST? **The tree that traces a parser derivation**
- ▶ What are the inner nodes of a CST?

# Concrete syntax tree

- ▶ What is a CST? **The tree that traces a parser derivation**
- ▶ What are the inner nodes of a CST? **The non-terminals**

# Concrete syntax tree

- ▶ What is a CST? **The tree that traces a parser derivation**
- ▶ What are the inner nodes of a CST? **The non-terminals**
- ▶ What are the leaves of a CST?

# Concrete syntax tree

- ▶ What is a CST? **The tree that traces a parser derivation**
- ▶ What are the inner nodes of a CST? **The non-terminals**
- ▶ What are the leaves of a CST? **The terminals**

# Abstract syntax tree

- ▶ What is a AST?

# Abstract syntax tree

- ▶ **What is a AST? A tree representation of the program without the extraneous stuff (e.g. punctuation, extra non-terminals)**

# Abstract syntax tree

- ▶ **What is a AST? A tree representation of the program without the extraneous stuff (e.g. punctuation, extra non-terminals)**
- ▶ **What are the inner nodes of an AST?**



# Abstract syntax tree

- ▶ What is a AST? **A tree representation of the program without the extraneous stuff (e.g. punctuation, extra non-terminals)**
- ▶ What are the inner nodes of an AST? **Statements and expressions**

# Abstract syntax tree

- ▶ What is a AST? **A tree representation of the program without the extraneous stuff (e.g. punctuation, extra non-terminals)**
- ▶ What are the inner nodes of an AST? **Statements and expressions**
- ▶ What are the leaves of an AST?

# Abstract syntax tree

- ▶ What is a AST? **A tree representation of the program without the extraneous stuff (e.g. punctuation, extra non-terminals)**
- ▶ What are the inner nodes of an AST? **Statements and expressions**
- ▶ What are the leaves of an AST? **Literals and identifiers**

# AST vs CST

- ▶ Can you use a CST for type checking?

# AST vs CST

- ▶ Can you use a CST for type checking? **Yes**

# AST vs CST

- ▶ Can you use a CST for type checking? **Yes**
- ▶ Can you use a CST for code gen?

# AST vs CST

- ▶ Can you use a CST for type checking? **Yes**
- ▶ Can you use a CST for code gen? **Yes**

# AST vs CST

- ▶ Can you use a CST for type checking? **Yes**
- ▶ Can you use a CST for code gen? **Yes**
- ▶ Then why do we prefer ASTs?



# AST vs CST

- ▶ Can you use a CST for type checking? **Yes**
- ▶ Can you use a CST for code gen? **Yes**
- ▶ Then why do we prefer ASTs? **Simpler and shorter**

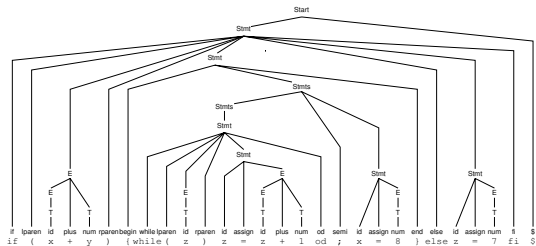


Figure 7.18: Concrete syntax tree.

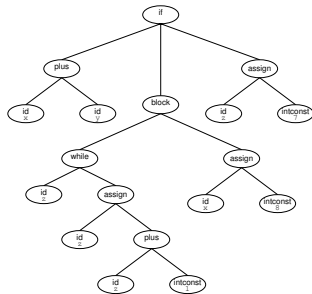


Figure 7.19: AST for the parse tree in Figure 7.18.

Weeder

# Weeder

What is the role of the weeder?

# Weeder

What is the role of the weeder?

**Reject invalid programs that the parser cannot.**

# Weeder

What are some examples that a parser cannot reject and must be done in a weeder?

# Weeder

What are some examples that a parser cannot reject and must be done in a weeder?

- ▶ Reject `break` and `continue` outside of loops
- ▶ Reject `switch` statements with multiple `default` branches
- ▶ Reject non-void functions without `return` statements

# Weeder

Can we write a parser to reject break outside loops?



# Weeder

Can we write a parser to reject break outside loops?

Probably, but the parser would be larger, more complicated and uglier.

# Weeder

If a check can be done in the parser and in the weeder, where should we do it?

# Weeder

If a check can be done in the parser and in the weeder, where should we do it?

- ▶ Where it makes our job easier
- ▶ Where it gives the better error message

# Symbol tables

# Symbol tables

What is stored in a symbol table?

# Symbol tables

What is stored in a symbol table?

**Identifiers and their related information.**

# Symbol tables

What information can be associated with a symbol?

# Symbol tables

What information can be associated with a symbol?

- ▶ Type
- ▶ Offset in stack frame
- ▶ Resources for methods (e.g. number of locals, stack limit)
- ▶ Original name
- ▶ Etc.



# Symbol tables

What data structure is typically used for symbol tables?

# Symbol tables

What data structure is typically used for symbol tables?

**Hash tables**

# Symbol tables

How do we handle multiple scopes where variables can be redeclared?

# Symbol tables

How do we handle multiple scopes where variables can be redeclared?

**Stack of hash tables**

# Symbol tables

How do we lookup a symbol?

# Symbol tables

How do we lookup a symbol?

**Search hash tables in the stack from top to bottom**

# Type checking

# Type checking

What is the role of type checking?



# Type checking

What is the role of type checking?

Reject programs that are *syntactically correct*, but *semantically wrong*.

# Type checking

- ▶ What is the input of the type checker?

# Type checking

- ▶ What is the input of the type checker? **AST**

# Type checking

- ▶ What is the input of the type checker? **AST**
- ▶ What is the output of the type checker?

# Type checking

- ▶ What is the input of the type checker? **AST**
- ▶ What is the output of the type checker? **Annotated AST**

# Type checking

- ▶ Do declarations have a type?

# Type checking

- ▶ Do declarations have a type? **No**

# Type checking

- ▶ Do declarations have a type? **No**
- ▶ Do statements have a type?



# Type checking

- ▶ Do declarations have a type? **No**
- ▶ Do statements have a type? **No**

# Type checking

- ▶ Do declarations have a type? **No**
- ▶ Do statements have a type? **No**
- ▶ Do expressions have a type?

# Type checking

- ▶ Do declarations have a type? **No**
- ▶ Do statements have a type? **No**
- ▶ Do expressions have a type? **Yes**

# Type checking

Where do we store the type of expressions?

# Type checking

Where do we store the type of expressions?

- ▶ In the AST
- ▶ In an auxiliary table (SableCC)

# Type checking

Exercise

```
var x int = expr
```

# Type checking

Exercise

```
var x int = expr
```

- ▶ Type check *expr*

# Type checking

## Exercise

```
var x int = expr
```

- ▶ Type check *expr*
- ▶ Make sure `int = typeof(expr)`



# Type checking

## Exercise

```
var x int = expr
```

- ▶ Type check *expr*
- ▶ Make sure `int = typeof(expr)`
- ▶ Report an error if the types don't match

# Type checking

## Exercise

```
var x int = expr
```

- ▶ Type check *expr*
- ▶ Make sure `int = typeof(expr)`
- ▶ Report an error if the types don't match
- ▶ Try to add `x -> int` to the symbol table

# Type checking

## Exercise

```
var x int = expr
```

- ▶ Type check *expr*
- ▶ Make sure `int = typeof(expr)`
- ▶ Report an error if the types don't match
- ▶ Try to add `x -> int` to the symbol table
- ▶ Report an error if `x` is already defined in the current scope

# Type checking

## Exercise

```
if expr {  
    then_stmts  
} else {  
    else_stmts  
}
```

# Type checking

## Exercise

```
if expr {  
    then_stmts  
} else {  
    else_stmts  
}
```

- ▶ Type check *expr*, *then\_stmts*, and *else\_stmts*

# Type checking

## Exercise

```
if expr {  
    then_stmts  
} else {  
    else_stmts  
}
```

- ▶ Type check *expr*, *then\_stmts*, and *else\_stmts*
- ▶ Make sure `typeof(expr) = bool`

# Type checking

## Exercise

```
if expr {  
    then_stmts  
} else {  
    else_stmts  
}
```

- ▶ Type check *expr*, *then\_stmts*, and *else\_stmts*
- ▶ Make sure `typeof(expr) = bool`
- ▶ Report an error if the types don't match

# Type checking

## Exercise

```
// x is declared as an int  
max(2+3, x)
```



# Type checking

## Exercise

```
// x is declared as an int  
max(2+3, x)
```

- ▶ Type check `2+3`
- ▶ Type check `x`
- ▶ Type check `max`
- ▶ Make sure `max` accepts two parameters and that `2+3` has the type of the first formal parameter and `x` has the type of the second formal parameter
- ▶ The whole expression has the return type declared for `max`

# Inference rules

What does this mean in English?

$$\frac{P}{C}$$

# Inference rules

What does this mean in English?

$$\frac{P}{C}$$

“If  $P$  then  $C$ ”

# Inference rules

What about this?

$$\frac{P_1 \quad P_2}{C}$$

# Inference rules

What about this?

$$\frac{P_1 \quad P_2}{C}$$

“If  $P_1$  and  $P_2$  then  $C$ ”

# Inference rules

What about this?

$$\frac{P_1 \quad P_2}{C}$$

“If  $P_1$  and  $P_2$  then  $C$ ”

Short version for:

$$\frac{P_1 \wedge P_2}{C}$$

# Inference rules

What does this mean in English?

$$\Gamma \vdash e : T$$

# Inference rules

What does this mean in English?

$$\Gamma \vdash e : T$$

“Under the set of assumptions  $\Gamma$ , *it is provable* ( $\vdash$ ) that  $e$  has type ( $:$ )  $T$ ”

(Assumptions = symbol table)



# Inference rules

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

# Inference rules

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

“If under the set of assumptions  $\Gamma$  it is provable that  $e_1$  has type *int* and under the set of assumptions  $\Gamma$  it is provable that  $e_2$  has type *int*, then under the set of assumptions  $\Gamma$  it is provable that  $e_1 + e_2$  has type *int*.”

# Inference rules

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \mathit{if } e \{s_1\} \mathit{else } \{s_2\}}$$

# Inference rules

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \mathit{if } e \{s_1\} \mathit{else } \{s_2\}}$$

“If under the set of assumptions  $\Gamma$  it is provable that  $e$  has type  $\mathit{bool}$  and under the set of assumptions  $\Gamma$  it is provable that  $s_1$  typechecks, and under the set of assumptions  $\Gamma$  it is provable that  $s_2$  typechecks, then under the set of assumptions  $\Gamma$  it is provable that  $\mathit{if } e \{s_1\} \mathit{else } \{s_2\}$  typechecks.”

# Inference rules

This is not going to be on the exam (probably)

$$\frac{\begin{array}{l} L, C, M, V \vdash E_i : \sigma_i \\ \exists \vec{\tau} : \text{constructor}(L, C, \vec{\tau}) \wedge \\ \quad \vec{\tau} := \vec{\sigma} \wedge \\ \quad (\forall \vec{\gamma} : \text{constructor}(L, C, \vec{\gamma}) \wedge \vec{\gamma} := \vec{\sigma} \\ \quad \quad \downarrow \\ \quad \quad \vec{\gamma} := \vec{\tau} \\ \quad ) \end{array}}{L, C, M, V \vdash \text{new } C(E_1, \dots, E_n) : C}$$

# Type derivation Grammar

```
expr = Id(x)  
      | Int(n)
```

```
stmt = 'var' Id type '=' expr ';' stmt  
      | 'print' expr ';' stmt  
      |  $\epsilon$ 
```

# Type derivation

## Type rules

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \textit{Id}(x) \quad \frac{}{\Gamma \vdash n : \textit{int}} \textit{Int}(n)$$

$$\frac{\Gamma \vdash e : T \quad (\Gamma, x : T) \vdash s}{\Gamma \vdash \textit{var } x T = e; s} \textit{var} \quad \frac{\Gamma \vdash e : T \quad \Gamma \vdash s}{\Gamma \vdash \textit{print } e; s} \textit{print} \quad \frac{}{\Gamma \vdash \epsilon} \textit{empty}$$

# Type derivation

`var z int = 4; print z; ε`

$$\frac{\frac{\frac{}{\{\} \vdash 4 : int} \text{Int} \quad \frac{\frac{\frac{\{z : int\}(z) = int}{\{z : int\} \vdash z : int} \text{Id}}{\{z : int\} \vdash print z; \epsilon} \text{print}}{\{z : int\} \vdash \epsilon} \text{empty}}{\{z : int\} \vdash print z; \epsilon} \text{var}}{\{\} \vdash var z int = 4; print z; \epsilon} \text{var}$$



# Code generation

# Code generation

Code generation has many sub-phases:

- ▶ Computing resources
- ▶ Generating an IR of the code
- ▶ Optimizing the code
- ▶ Emitting the code

# Computing resources

In JOOS, what resources did we need to compute?

- ▶ L
- ▶ S
- ▶ L
- ▶ O

# Computing resources

In JOOS, what resources did we need to compute?

- ▶ Locals (how many?)
- ▶ S
- ▶ L
- ▶ O

# Computing resources

In JOOS, what resources did we need to compute?

- ▶ Locals (how many?)
- ▶ Stack height (maximum)
- ▶ L
- ▶ O

# Computing resources

In JOOS, what resources did we need to compute?

- ▶ Locals (how many?)
- ▶ Stack height (maximum)
- ▶ Labels (for control structures and some operators)
- ▶ O

# Computing resources

In JOOS, what resources did we need to compute?

- ▶ Locals (how many?)
- ▶ Stack height (maximum)
- ▶ Labels (for control structures and some operators)
- ▶ Offsets (locals and formals)

# IR

Which IRs did we see in class?



# IR

Which IRs did we see in class?

**JVM Bytecodes and VirtualRISC**

# JVM bytecodes

What does the body of this method look like in Jasmin?

```
public static void f(int x) {  
    x = x + 3;  
}
```

# JVM bytecodes

What does the body of this method look like in Jasmin?

```
public static void f(int x) {  
    x = x + 3;  
}
```

```
                                // [ TOP , BOT ]  
                                // [   ,   ]  
iload_0                        // [ x   ,   ]  
ldc_int 3                      // [ 3   , x   ]  
iadd                           // [ x+3 ,   ]  
istore_0                       // [   ,   ]
```

# JVM bytecodes

What does the body of this method look like in Jasmin?

```
public static void f(int x) {  
    x = x + 3;  
}
```

```
                                // [ TOP , BOT ]  
                                // [   ,   ]  
iload_0                        // [ x   ,   ]  
ldc_int 3                      // [ 3   , x   ]  
iadd                          // [ x+3 ,   ]  
istore_0                       // [   ,   ]
```

- ▶ How many locals?

# JVM bytecodes

What does the body of this method look like in Jasmin?

```
public static void f(int x) {  
    x = x + 3;  
}
```

```
                                // [ TOP , BOT ]  
                                // [      ,      ]  
iload_0                          // [ x   ,      ]  
ldc_int 3                        // [ 3   , x   ]  
iadd                             // [ x+3 ,      ]  
istore_0                         // [      ,      ]
```

- ▶ How many locals? **1**

# JVM bytecodes

What does the body of this method look like in Jasmin?

```
public static void f(int x) {  
    x = x + 3;  
}
```

```
                                // [ TOP , BOT ]  
                                // [      ,      ]  
iload_0                        // [ x   ,      ]  
ldc_int 3                      // [ 3   , x   ]  
iadd                          // [ x+3 ,      ]  
istore_0                      // [      ,      ]
```

- ▶ How many locals? **1**
- ▶ Stack height?

# JVM bytecodes

What does the body of this method look like in Jasmin?

```
public static void f(int x) {  
    x = x + 3;  
}
```

```
                                // [ TOP , BOT ]  
                                // [      ,      ]  
iload_0                        // [ x   ,      ]  
ldc_int 3                      // [ 3   , x   ]  
iadd                           // [ x+3 ,      ]  
istore_0                       // [      ,      ]
```

- ▶ How many locals? **1**
- ▶ Stack height? **2**

# JVM bytecodes

How would we generate code for the following pattern?

```
if (E) S1 else S2
```



# JVM bytecodes

How would we generate code for the following pattern?

```
if (E) S1 else S2
```

```
<code for E>  
ifeq else_branch  
<code for S1>  
goto end_if  
else_branch:  
<code for S2>  
end_if:
```

# JVM bytecodes

What invariant must be respected by *statement* code templates?

# JVM bytecodes

What invariant must be respected by *statement* code templates?

**Stack height is unchanged**

# JVM bytecodes

What invariant must be respected by *statement* code templates?

**Stack height is unchanged**

What invariant must be respected by *expression* code templates?

# JVM bytecodes

What invariant must be respected by *statement* code templates?

**Stack height is unchanged**

What invariant must be respected by *expression* code templates?

**Stack height increased by one**

# VirtualRISC

VirtualRISC is what type of machine?

# VirtualRISC

VirtualRISC is what type of machine?

**RISC machine**

# VirtualRISC

VirtualRISC is what type of machine?

**RISC machine**

What are its registers?



# VirtualRISC

VirtualRISC is what type of machine?

**RISC machine**

What are its registers?

**Unbounded number of general purpose registers, *sp* stack pointer, *fp* frame pointer, and *pc* program counter**

# VirtualRISC

What does its stack frame look like?

# VirtualRISC

What does its stack frame look like?

