Tentative syllabus

Front End
- Scanning
- Top-down Parsing (LL)
- Bottom-up Parsing (LR)

Intermediate Representation
- Operational Semantics
- Lowering

Optimizations
- Dataflow Analysis
- Loop Optimizations

Code Generation
- Register Allocation
- Instruction Selection

mid-term

exam

2
PREVIOUSLY

• Becoming parsing ninjas
  – Going from text to an Abstract Syntax Tree
From scanning to parsing

program text

59 + (1257 * xPosition)

Lexical Analyzer

num + ( num * id )

Parser

Grammar:
E → id
E → num
E → E + E
E → E * E
E → ( E )

Abstract Syntax Tree
The role of intermediate representations

Two example languages
  - A high-level language
  - An intermediate language

Lowering

Correctness
  - Formal meaning of programs
Role of intermediate representation

• Bridge between front-end and back-end

High-level Language (scheme)  Lexical Analysis  Syntax Analysis Parsing  AST  Symbol Table etc.  Inter. Rep. (IR)  Code Generation  Executable Code

• Allow implementing optimizations independent of source language and executable (target) language
MOTIVATION FOR INTERMEDIATE REPRESENTATION
Intermediate representation

• A language that is between the source language and the target language
  – Not specific to any source language or machine language

• Goal 1: retargeting compiler components for different source languages/target machines

Diagram:

- C++ → IR
- Java → IR
- Python → IR
- Pentium → IR
- Java bytecode → IR
- Sparc → IR
Intermediate representation

• A language that is between the source language and the target language
  – Not specific to any source language or machine language

• Goal 1: retargeting compiler components for different source languages/target machines

• Goal 2: machine-independent optimizer
  – Narrow interface: small number of node types (instructions)
Multiple IRs

• Some optimizations require high-level constructs while others more appropriate on low-level code

• Solution: use multiple IR stages

\[ \text{AST} \rightarrow \text{HIR} \rightarrow \text{LIR} \]

- optimize
- optimize

- Pentium
- Java bytecode
- Sparc
Multiple IRs example

Elixir – a language for parallel graph algorithms

Elixir Program

Delta Inferencer → Elixir Program + delta → HIR Lowering → HIR

Query → Answer

Automated Reasoning (Boogie+Z3)

IL Synthesizer

Planning Problem → Plan → Automated Planner

LIR

C++ backend

C++ code

Galois Library
## AST vs. LIR for imperative languages

<table>
<thead>
<tr>
<th>AST</th>
<th>LIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Rich set of language constructs</td>
<td>• An abstract machine language</td>
</tr>
<tr>
<td>• Rich type system</td>
<td>• Very limited type system</td>
</tr>
<tr>
<td>• Declarations: types (classes, interfaces), functions, variables</td>
<td>• Only computation-related code</td>
</tr>
<tr>
<td>• Control flow statements: if-then-else, while-do, break-continue, switch, exceptions</td>
<td>• Labels and conditional/unconditional jumps, no looping</td>
</tr>
<tr>
<td>• Data statements: assignments, array access, field access</td>
<td>• Data movements, generic memory access statements</td>
</tr>
<tr>
<td>• Expressions: variables, constants, arithmetic operators, logical operators, function calls</td>
<td>• No sub-expressions, logical as numeric, temporaries, constants, function calls – explicit argument passing</td>
</tr>
</tbody>
</table>
THREE ADDRESS CODE
Three-Address Code IR

• A popular form of IR
• High-level assembly where instructions have at most three operands

• There exist other types of IR
  – For example, IR based on acyclic graphs – more amenable for analysis and optimizations
BASE LANGUAGE: While
## Syntax

\[ n \in \textbf{Num} \quad \text{numerals} \]
\[ x \in \textbf{Var} \quad \text{program variables} \]

\[ A \rightarrow n \mid x \mid A \ \text{ArithOp} \ A \mid (A) \]
\[ \text{ArithOp} \rightarrow - \mid + \mid * \]
\[ B \rightarrow \text{true} \mid \text{false} \]
\[ \mid A = A \mid A \leq A \mid \neg B \mid B \land B \mid (B) \]
\[ S \rightarrow x := A \mid \text{skip} \mid S; S \mid \{ S \} \]
\[ \mid \text{if } B \text{ then } S \text{ else } S \]
\[ \mid \text{while } B \ S \]
Example program

```plaintext
while x < y {
    x := x + 1
}

y := x;
```
INTERMEDIATE LANGUAGE:

IL
## Syntax

<table>
<thead>
<tr>
<th>Grammar Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n \in \text{Num} )</td>
<td>Numerals</td>
</tr>
<tr>
<td>( l \in \text{Num} )</td>
<td>Labels</td>
</tr>
<tr>
<td>( x \in \text{Temp} \cup \text{Var} )</td>
<td>Temporaries and variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V \rightarrow n \mid x )</td>
<td></td>
</tr>
<tr>
<td>( R \rightarrow V , \text{Op} , V )</td>
<td></td>
</tr>
<tr>
<td>( \text{Op} \rightarrow - \mid + \mid \ast \mid = \mid \leq \mid \ll \mid \gg \mid \ldots )</td>
<td></td>
</tr>
<tr>
<td>( C \rightarrow l: \text{skip} )</td>
<td></td>
</tr>
<tr>
<td>( l: x := R )</td>
<td></td>
</tr>
<tr>
<td>( l: \text{Goto} , l' )</td>
<td></td>
</tr>
<tr>
<td>( l: \text{IfZ} , x , \text{Goto} , l' )</td>
<td></td>
</tr>
<tr>
<td>( l: \text{IfNZ} , x , \text{Goto} , l' )</td>
<td></td>
</tr>
<tr>
<td>( \text{IR} \rightarrow C^+ )</td>
<td></td>
</tr>
</tbody>
</table>
Intermediate language programs

• An intermediate program $P$ has the form
  1:c_1
  ...
  n:c_n

• We can view it as a map from labels to individual commands and write $P(j) = c_j$

```plaintext
1: t0 := 137
2: y := t0 + 3
3: IfZ x Goto 7
4: t1 := y
5: z := t1
6: Goto 9
7: t2 := y
8: x := t2
9: skip
```
LOWERING
TAC generation

• At this stage in compilation, we have
  – an AST
  – annotated with scope information
  – and annotated with type information

• We generate TAC recursively (bottom-up)
  – First for expressions
  – Then for statements
TAC generation for expressions

• Define a function \texttt{cgen(expr)} that generates TAC that:
  1. Computes an expression,
  2. Stores it in a temporary variable,
  3. Returns the name of that temporary

• Define \texttt{cgen} directly for atomic expressions (constants, this, identifiers, etc.)

• Define \texttt{cgen} recursively for compound expressions (binary operators, function calls, etc.)
Translation rules for expressions

\[\text{cgen}(n) = (l: t:=n, t) \quad \text{where } l \text{ and } t \text{ are fresh}\]

\[\text{cgen}(x) = (l: t:=x, t) \quad \text{where } l \text{ and } t \text{ are fresh}\]

\[\begin{align*}
\text{cgen}(e_1) &= (P_1, t_1) \\
\text{cgen}(e_2) &= (P_2, t_2) \\
\text{cgen}(e_1 \text{ op } e_2) &= (P_1 \cdot P_2 \cdot (l: t:=t_1 \text{ op } t_2, t)) \\
\end{align*}\]

\[\text{where } l \text{ and } t \text{ are fresh}\]
cgen for basic expressions

- Maintain a counter for temporaries in c, and a counter for labels in l
- Initially: c = 0, l = 0

```cgen(k) = { // k is a constant
    c = c + 1, l = l + 1
    emit(l: tc := k)
    return tc
}

cgen(id) = { // id is an identifier
    c = c + 1, l = l + 1
    emit(l: tc := id)
    return tc
}
Naive \texttt{cgen} for binary expressions

\begin{itemize}
  \item \texttt{cgen}(e_1 \; op \; e_2) = \{
    \begin{align*}
    \text{let } A &= \texttt{cgen}(e_1) \\
    \text{let } B &= \texttt{cgen}(e_2) \\
    c &= c + 1, \; l = l + 1
    \end{align*}
    \text{emit}( l: \texttt{tc} := A \; op \; B; ) \\
    \text{return } \texttt{tc}
  \}
\end{itemize}

The translation emits code to evaluate \( e_1 \) before \( e_2 \). Why is that?
Example: `cgen` for binary expressions

`cgen( (a*b)-d)`
Example: \texttt{cgen} for binary expressions

\[ c = 0, \; l = 0 \]
\[ \texttt{cgen}( (a*b)-d) \]
Example: \texttt{cgen} for binary expressions

c = 0, l = 0
\texttt{cgen}( (a*b)-d) = 
\begin{align*}
&\text{let } A = \texttt{cgen}(a*b) \\
&\text{let } B = \texttt{cgen}(d) \\
&c = c + 1, l = l + 1 \\
&\text{emit}(l: tc := A - B; ) \\
&\text{return } tc \\
\end{align*}
Example: \texttt{cgen} for binary expressions

\begin{verbatim}
c = 0, l = 0
cgen( (a*b)-d) = {
  let A = {
    let A = cgen(a)
    let B = cgen(b)
    c = c + 1, l = l +1
    emit(l: tc := A * B; )
    return tc
  }
  let B = cgen(d)
  c = c + 1, l = l +1
  emit(l: tc := A - B; )
  return tc
}
\end{verbatim}
Example: \texttt{cgen} for binary expressions

\begin{verbatim}
c = 0, l = 0
cgen( (a*b)-d) = {
    let A = {
        let A = {c=c+1, l=l+1, emit(l: tc := a;), return tc }
        let B = {c=c+1, l=l+1, emit(l: tc := b;), return tc }
        c = c + 1, l = l + 1
        emit(l: tc := A * B; )
        return tc
    }
    let B = {c=c+1, l=l+1, emit(l: tc := d;), return tc }
    c = c + 1, l = l + 1
    emit(l: tc := A - B; )
    return tc
}
\end{verbatim}
Example: \texttt{cgen} for binary expressions

\begin{verbatim}
c = 0, l = 0
\texttt{cgen}((a*b)-d) = {
  let A = {
    let A = {c=c+1, l=l+1, emit(l: tc := a;), return tc }
    let B = {c=c+1, l=l+1, emit(l: tc := b;), return tc }
    c = c + 1, l = l + 1
    emit(l: tc := A * B; )
    return tc
  }
  let B = {c=c+1, l=l+1, emit(l: tc := d;), return tc }
  c = c + 1, l = l + 1
  emit(l: tc := A - B; )
  return tc
}
\end{verbatim}
Example: \texttt{cgen} for binary expressions

\begin{align*}
c &= 0, \quad l = 0 \\
c\text{gen}( (a*b)-d) &= \{
\text{let } A = \{
\text{let } A = \{c=c+1, \quad l=l+1, \quad \text{emit}(l: \quad tc := a;), \quad \text{return } tc \} \\
\text{let } B = \{c=c+1, \quad l=l+1, \quad \text{emit}(l: \quad tc := b;), \quad \text{return } tc \} \\
c &= c + 1, \quad l = l + 1 \\
\text{emit}(l: \quad tc := A * B; ) \\
\text{return } tc
\} \\
\text{let } B = \{c=c+1, \quad l=l+1, \quad \text{emit}(l: \quad tc := d;), \quad \text{return } tc \} \\
c &= c + 1, \quad l = l + 1 \\
\text{emit}(l: \quad tc := A - B; ) \\
\text{return } tc
\}
\end{align*}

Code
\begin{align*}
1: & \quad t1 := a; \\
2: & \quad t2 := b;
\end{align*}
Example: cgen for binary expressions

c = 0, l = 0

cgen( (a*b)-d) = {
    let A = {
        let A = {c=c+1, l=l+1, emit(l: tc := a;), return tc }
        let B = {c=c+1, l=l+1, emit(l: tc := b;), return tc }
        c = c + 1, l = l + 1
        emit(l: tc := A * B; )
        return tc
    }
    let B = {c=c+1, l=l+1, emit(l: tc := d;), return tc }
    c = c + 1, l = l + 1
    emit(l: tc := A - B; )
    return tc
}
Example: \texttt{cgen} for binary expressions

\begin{align*}
c &= 0, \quad l = 0 \\
cgen((a*b) - d) &= \\
\text{let } A &= \\
\quad \text{let } A = \\
\quad \quad c = c + 1, \quad l = l + 1, \quad \text{emit}(l: tc := a;), \quad \text{return } tc \\
\quad \text{let } B = \\
\quad \quad c = c + 1, \quad l = l + 1, \quad \text{emit}(l: tc := b;), \quad \text{return } tc \\
\quad c = c + 1, \quad l = l + 1 \\
\quad \text{emit}(l: tc := A \cdot B; ) \\
\quad \text{return } tc \\
\end{align*}

\begin{align*}
\text{Code} \\
1: & \quad t1 := a; \\
2: & \quad t2 := b; \\
3: & \quad t3 := t1 \cdot t2 \\
\end{align*}
Example: \texttt{cgen} for binary expressions

\begin{align*}
c = 0, \ l = 0 \\
c \text{gen}( (a*b) - d) = \{ \\
\text{let} \ A = \{ \\
\quad \text{let} \ A = \{ c = c + 1, \ l = l + 1, \ \text{emit}(l: \ tc := a);), \ \text{return} \ tc \} \\
\quad \text{let} \ B = \{ c = c + 1, \ l = l + 1, \ \text{emit}(l: \ tc := b);), \ \text{return} \ tc \} \\
\quad c = c + 1, \ l = l + 1 \\
\quad \text{emit}(l: \ tc := A * B; ) \\
\quad \text{return} \ tc \\
\} \\
\text{let} \ B = \{ c = c + 1, \ l = l + 1, \ \text{emit}(l: \ tc := d);), \ \text{return} \ tc \} \\
\quad c = c + 1, \ l = l + 1 \\
\quad \text{emit}(l: \ tc := A - B; ) \\
\quad \text{return} \ tc \\
\}
\end{align*}
Example: \texttt{cgen} for binary expressions

c = 0, l = 0
\texttt{cgen( (a*b) - d) = { }
\texttt{let A = { }
\texttt{let A = \{c=c+1, l=l+1, emit(l: tc := a;), return tc \}}
\texttt{let B = \{c=c+1, l=l+1, emit(l: tc := b;), return tc \}}
\texttt{c = c + 1, l = l + 1}
\texttt{emit(l: tc := A * B; )}
\texttt{return tc }
\texttt{}}
\texttt{let B = \{c=c+1, l=l+1, emit(l: tc := d;), return tc \}}
\texttt{c = c + 1, l = l + 1}
\texttt{emit(l: tc := A - B; )}
\texttt{return tc }
\texttt{}}

Code
1: t1:=a;
2: t2:=b;
3: t3:=t1*t2
4: t4:=d
5: t5:=t3-t4
cgen for statements

• We can extend the cgen function to operate over statements as well

• Unlike cgen for expressions, cgen for statements does not return the name of a temporary holding a value
  – (Why?)
# Syntax

$n \in \textbf{Num}$ numerals

$x \in \textbf{Var}$ program variables

\[
A \rightarrow n \mid x \mid A \text{ ArithOp } A \mid (A)
\]

\[
\text{ArithOp} \rightarrow - \mid + \mid * \mid /
\]

\[
B \rightarrow \text{true} \mid \text{false}
\]

\[
\mid A = A \mid A \leq A \mid \neg B \mid B \land B \mid (B)
\]

\[
S \rightarrow x := A \mid \text{skip} \mid S; S \mid \{ \ S \} \\
\mid \text{if } B \text{ then } S \text{ else } S \\
\mid \text{while } B \ S
\]
Translation rules for statements

\[
\text{cgen}(\textbf{skip}) = l: \text{skip} \quad \text{where } l \text{ is fresh}
\]

\[
\begin{array}{l}
\text{cgen}(e) = (P, t) \\
\text{cgen}(x := e) = P \cdot l: x := t
\end{array}
\quad \text{where } l \text{ is fresh}
\]

\[
\begin{array}{l}
\text{cgen}(S_1) = P_1, \quad \text{cgen}(S_2) = P_2 \\
\text{cgen}(S_1; S_2) = P_1 \cdot P_2
\end{array}
\]
Translation rules for conditions

\[
\text{cgen}(b) = (Pb, t), \quad \text{cgen}(S_1) = P_1, \quad \text{cgen}(S_2) = P_2
\]

\[
\text{cgen}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2) =
\]

\[
Pb
\]

\[
\text{IfZ } t \text{ Goto } l_{\text{false}}
\]

\[
P_1
\]

\[
l_{\text{finish}} : \text{Goto } L_{\text{after}}
\]

\[
l_{\text{false}} : \text{skip}
\]

\[
P_2
\]

\[
l_{\text{after}} : \text{skip}
\]

where \(l_{\text{finish}}, l_{\text{false}}, l_{\text{after}}\) are fresh.
Translation example

\[
\begin{align*}
y & := 137 + 3; \\
\text{if } x & = 0 \\
& \quad z := y; \\
\text{else} \\
& \quad x := y; \\
\end{align*}
\]

1: \( t1 := 137 \)
2: \( t2 := 3 \)
3: \( t3 := t1 + t2 \)
4: \( y := t3 \)
5: \( t4 := x \)
6: \( t5 := 0 \)
7: \( t6 := t4 = t5 \)
8: \( \text{IfZ } t6 \text{ Goto 12} \)
9: \( t7 := y \)
10: \( z := t7 \)
11: \( \text{Goto 14} \)
12: \( t8 := y \)
13: \( x := t8 \)
14: \( \text{skip} \)
Translation rule for loops

\[ \text{cgen}(b) = (Pb, t), \quad \text{cgen}(S) = P \]

\[ \text{cgen}(\text{while } b \ S) = \]

\[ \begin{align*}
\text{l}_{\text{before}} & : \text{skip} \\
Pb & \\
\text{IfZ } t \ \text{Goto } \text{l}_{\text{after}} & \\
P & \\
\text{l}_{\text{loop}} & : \text{Goto } \text{l}_{\text{before}} \\
\text{l}_{\text{after}} & : \text{skip}
\end{align*} \]

where \( \text{l}_{\text{after}}, \text{l}_{\text{before}}, \text{l}_{\text{loop}} \)
are fresh
Translation example

\begin{align*}
y &:= 137+3; \\
\text{while } &x=0 \\
&z := y;
\end{align*}

\begin{align*}
1: &\ t1 := 137 \\
2: &\ t2 := 3 \\
3: &\ t3 := t1 + t2 \\
4: &\ y := t3 \\
5: &\ t4 := x \\
6: &\ t5 := 0 \\
7: &\ t6 := t4=t5 \\
8: &\ IfZ t6 Goto 12 \\
9: &\ t7 := y \\
10: &\ z := t7 \\
11: &\ Goto 14 \\
12: &\ skip
\end{align*}
TRANSLATION CORRECTNESS
Compiler correctness

• Compilers translate programs in one language (usually high) to another language (usually lower) such that they are both equivalent

• Our goal is to formally define the meaning of this equivalence
  – First, we must formally define the meaning of programs
What is formal semantics?

“Formal semantics is concerned with rigorously specifying the meaning, or behavior, of programs, pieces of hardware, etc.”
Why formal semantics?

• Implementation-independent definition of a programming language

• Automatically generating interpreters (and some day maybe full fledged compilers)

• Optimization, verification, and debugging
  – If you don’t know what it does, how do you know its correct/incorrect?
  – How do you know whether a given optimization is correct?
Operational semantics

• Elements of the semantics
• **States/configurations**: the (aggregate) values that a program computes during execution
• **Transition rules**: how the program advances from one configuration to another
OPERATIONAL SEMANTICS OF WHILE
While syntax reminder

\[ n \in \text{Num} \quad \text{numerals} \]
\[ x \in \text{Var} \quad \text{program variables} \]

\[ A \rightarrow n \mid x \mid A \text{ ArithOp } A \mid (A) \]
\[ \text{ArithOp} \rightarrow - \mid + \mid * \]
\[ B \rightarrow \text{true} \mid \text{false} \]
\[ \mid A = A \mid A \leq A \mid \neg B \mid B \land B \mid (B) \]
\[ S \rightarrow x := A \mid \text{skip} \mid S; S \mid \{ S \} \]
\[ \mid \text{if } B \text{ then } S \text{ else } S \]
\[ \mid \text{while } B \text{ do } S \]
Semantic categories

Z  Integers \{0, 1, -1, 2, -2, \ldots\}
T  Truth values \{ff, tt\}
State  Var \rightarrow Z

Example state:  \(\sigma = [x \mapsto 5, \ y \mapsto 7, \ z \mapsto 0]\)
Lookup:  \(\sigma(x) = 5\)
Update:  \(\sigma[x \mapsto 6] = [x \mapsto 6, \ y \mapsto 7, \ z \mapsto 0]\)
SEMANTICS OF EXPRESSIONS
Semantics of arithmetic expressions

• Semantic function $[A]: \text{State} \rightarrow \mathbb{Z}$

• Defined by induction on the syntax tree

\[
\begin{align*}
\llbracket n \rrbracket \sigma &= n \\
\llbracket x \rrbracket \sigma &= \sigma(x) \\
\llbracket a_1 + a_2 \rrbracket \sigma &= \llbracket a_1 \rrbracket \sigma + \llbracket a_2 \rrbracket \sigma \\
\llbracket a_1 - a_2 \rrbracket \sigma &= \llbracket a_1 \rrbracket \sigma - \llbracket a_2 \rrbracket \sigma \\
\llbracket a_1 \times a_2 \rrbracket \sigma &= \llbracket a_1 \rrbracket \sigma \times \llbracket a_2 \rrbracket \sigma \\
\llbracket (a_1) \rrbracket \sigma &= \llbracket a_1 \rrbracket \sigma \quad \text{--- not needed} \\
\llbracket - a \rrbracket \sigma &= 0 - \llbracket a_1 \rrbracket \sigma
\end{align*}
\]

• Compositional

• Expressions in While are side-effect free
Suppose $\sigma x = 3$

Evaluate $\left[ x + 1 \right] \sigma$
Semantics of boolean expressions

• Semantic function \([ B ] : \text{State} \rightarrow T\)

• Defined by induction on the syntax tree

  \[
  \begin{align*}
  [\text{true}] \sigma &= \text{tt} \\
  [\text{false}] \sigma &= \text{ff} \\
  [a_1 = a_2] \sigma &= \\
  [a_1 \leq a_2] \sigma &= \\
  [b_1 \land b_2] \sigma &= \\
  [\neg b] \sigma &= 
  \end{align*}
  \]

• Compositional

• Expressions in While are side-effect free
Semantics of statements

$$\langle S, \sigma \rangle \rightarrow \gamma$$

first step
Operational semantics

- Developed by Gordon Plotkin

- **Configurations**: $\gamma$ has one of two forms:
  
  $\langle S, \sigma \rangle$  
  Statement $S$ is about to execute on state $\sigma$

  $\sigma$  
  Terminal (final) state

- **Transitions** $\langle S, \sigma \rangle \rightarrow \gamma$
  
  $\gamma = \langle S', \sigma' \rangle$ Execution of $S$ from $\sigma$ is **not** completed and remaining computation proceeds from intermediate configuration $\gamma$

  $\gamma = \sigma'$ Execution of $S$ from $\sigma$ has **terminated** and the final state is $\sigma'$

- $\langle S, \sigma \rangle$ is **stuck** if there is no $\gamma$ such that $\langle S, \sigma \rangle \rightarrow \gamma$
Form of semantic rules

- Defined by rules of the form

$$\langle S_1, \sigma_1 \rangle \rightarrow \gamma_1', \ldots, \langle S_n, \sigma_n \rangle \rightarrow \gamma_n' \quad \text{if...}$$

- The meaning of compound statements is defined using the meaning immediate constituent statements
Operational semantics for **While**

\[
\begin{align*}
\textbf{[ass]} & \quad \langle x:=a, \sigma \rangle \rightarrow \sigma[x\mapsto [a] \sigma] \\
\textbf{[skip]} & \quad \langle \text{skip}, \sigma \rangle \rightarrow \sigma \\
\textbf{[comp]} & \quad \begin{array}{c}
\frac{\langle S_1, \sigma \rangle \rightarrow \langle S'_1, \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S'_1; S_2, \sigma' \rangle} \\
\frac{\langle S_1, \sigma \rangle \rightarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle}
\end{array}
\end{align*}
\]

\[\text{When does this happen?}\]

\[
\begin{align*}
\textbf{[if]} & \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle \quad \text{if } \llbracket b \rrbracket \sigma = \text{tt} \\
\textbf{[iff]} & \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle \quad \text{if } \llbracket b \rrbracket \sigma = \text{ff} \\
\textbf{[while]} & \quad \langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \langle S; \text{while } b \text{ do } S, \sigma \rangle \quad \text{if } \llbracket b \rrbracket \sigma = \text{tt} \\
\textbf{[whilef]} & \quad \langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma \quad \text{if } \llbracket b \rrbracket \sigma = \text{ff}
\end{align*}
\]
Factorial \((n!)\) example

- **Input state** \(\sigma = [x \mapsto 3, y \mapsto 0]\)

\[
y := 1; \text{ while } \neg (x=1) \text{ do } (y := y \times x; \ x := x - 1)
\]

- \(\langle y := 1; W, [x \mapsto 3, y \mapsto 0] \rangle\)
- \(\rightarrow \langle W, [x \mapsto 3, y \mapsto 1] \rangle\)
- \(\rightarrow \langle ((y := y \times x; x := x - 1); W), [x \mapsto 3, y \mapsto 1] \rangle\)
- \(\rightarrow \langle (x := x - 1; W), [x \mapsto 3, y \mapsto 3] \rangle\)
- \(\rightarrow \langle W, [x \mapsto 2, y \mapsto 3] \rangle\)
- \(\rightarrow \langle ((y := y \times x; x := x - 1); W), [x \mapsto 2, y \mapsto 3] \rangle\)
- \(\rightarrow \langle (x := x - 1; W), [x \mapsto 2, y \mapsto 6] \rangle\)
- \(\rightarrow \langle W, [x \mapsto 1, y \mapsto 6] \rangle\)
- \(\rightarrow [x \mapsto 1, y \mapsto 6]\)
Derivation sequences

• An interpreter
  – Given a statement $S$ and an input state $\sigma$
    start with the initial configuration $\langle S, \sigma \rangle$
  – Repeatedly apply rules until either a terminal state
    is reached or an infinite sequence

• We will write $\langle S, \sigma \rangle \rightarrow^k \sigma'$ if $\sigma'$ can be obtained
  from $\langle S, \sigma \rangle$ by $k$ derivation steps

• We will write $\langle S, \sigma \rangle \rightarrow^* \sigma'$ if $\langle S, \sigma \rangle \rightarrow^k \sigma'$ for
  some $k \geq 0$
The semantics of statements

• The meaning of a statement $S$ is defined as

$$\left[ S \right] \sigma = \begin{cases} \sigma' & \text{if } \langle S, \sigma \rangle \rightarrow^* \sigma' \\ \perp & \text{else} \end{cases}$$

• Examples:

$$\left[ \text{skip} \right] \sigma = \sigma$$
$$\left[ x := 1 \right] \sigma = \sigma[x \mapsto 1]$$
$$\left[ \text{while true do skip} \right] \sigma = \perp$$
PROPERTIES OF OPERATIONAL SEMANTICS
Deterministic semantics for While

• **Theorem:** for all statements \( S \) and states \( \sigma_1, \sigma_2 \)
  
  if \( \langle S, \sigma \rangle \rightarrow^* \sigma_1 \) and \( \langle S, \sigma \rangle \rightarrow^* \sigma_2 \) then \( \sigma_1 = \sigma_2 \)

• **Theorem:** for all statements \( S \) and states \( \sigma \), no derivation sequence from \( \langle S, \sigma \rangle \) ever reaches a stuck state
Program termination

• Given a statement $S$ and input $\sigma$
  
  – $S$ terminates on $\sigma$ if there exists a state $\sigma'$ such that $\langle S, \sigma \rangle \rightarrow^* \sigma'$
  
  – $S$ loops on $\sigma$ if there is no state $\sigma'$ such that $\langle S, \sigma \rangle \rightarrow^* \sigma'$

• Given a statement $S$
  
  – $S$ always terminates if
    for every input state $\sigma$, $S$ terminates on $\sigma$
  
  – $S$ always loops if
    for every input state $\sigma$, $S$ loops on $\sigma$
Semantic equivalence

- $S_1$ and $S_2$ are **semantically equivalent** if for all $\sigma$:
  \[
  \langle S_1, \sigma \rangle \rightarrow^* \gamma \text{ if and only if } \langle S_2, \sigma \rangle \rightarrow^* \gamma
  \]
  where $\gamma$ is either stuck or terminal and there is an infinite derivation sequence for
  \[
  \langle S_1, \sigma \rangle \text{ if and only if there is one for } \langle S_2, \sigma \rangle
  \]
  - A formal basis for correct optimizations of While programs

- **Examples**
  - $S; \text{skip}$ and $S$
  - $\text{while } b \text{ do } S \text{ and if } b \text{ then } (S; \text{while } b \ S) \text{ else } \text{skip}$
  - $((S_1; S_2); S_3) \text{ and } (S_1; (S_2; S_3))$
  - $(x:=5; \ y:=x*8) \text{ and } (x:=5; \ y:=40)$
OPERATIONAL SEMANTICS OF IL
IL syntax reminder

\[ n \in \textbf{Num} \quad \text{Numerals} \]
\[ l \in \textbf{Num} \quad \text{Labels} \]
\[ x \in \textbf{Var} \cup \textbf{Temp} \quad \text{Variables and temporaries} \]

\[
V \rightarrow n \mid x \\
R \rightarrow V \ Op \ V \mid V \\
Op \rightarrow - \mid + \mid * \mid = \mid \leq \mid \ll \mid \gg \mid \ldots \\
C \rightarrow l:\ \texttt{skip} \\
\mid l:\ x := R \\
\mid l:\ \texttt{Goto } l' \\
\mid l:\ \texttt{IfZ } x \\texttt{Goto } l' \\
IR \rightarrow C^+\]
Intermediate program states

\[ Z \text{ Integers } \{0, 1, -1, 2, -2, \ldots \} \]

\[ IState \quad (\text{Var} \cup \text{Temp} \cup \{\text{pc}\}) \rightarrow Z \]

- pc (for program counter) is a special variable
- \text{Var}, \text{Temp}, and \{pc\} are all disjoint
- For every intermediate program state \( m \) and program \( P=1:c_1,\ldots,n:c_n \) we have that \( 1 \leq m(\text{pc}) \leq n+1 \)
  - We can check that the labels used in \( P \) are legal
Rules for executing commands

• We will use rules of the following form

\[
\begin{array}{c}
m(pc) = l \quad P(l) = C \\
m \triangleright m'
\end{array}
\]

• Here \( m \) is the **pre-state**, which is scheduled to be executed as the program counter indicates, and \( m' \) is the **post-state**

• The rules specialize for the particular type of command \( C \) and possibly other conditions
Evaluating values

\[[n]m = n\]

\[[x]m = m(x)\]

\[[v_1 \op v_2]m = [v_1]m \op [v_2]m\]
Rules for executing commands

\[
\begin{align*}
  m(\text{pc}) &= l & P(l) &= \text{skip} \\
  m &\triangleright m[\text{pc}\mapsto l+1]
\end{align*}
\]

\[
\begin{align*}
  m(\text{pc}) &= l & P(l) &= x := v \\
  m &\triangleright m[\text{pc}\mapsto l+1, \ x\mapsto \lbrack v \rbrack \ m]
\end{align*}
\]

\[
\begin{align*}
  m(\text{pc}) &= l & P(l) &= x := v_1 \ op \ v_2 \\
  m &\triangleright m[\text{pc}\mapsto l+1, \ x\mapsto \lbrack v_1 \ op \ v_2 \rbrack \ m]
\end{align*}
\]

\[
\begin{align*}
  m(\text{pc}) &= l & P(l) &= \text{Goto} \ l' \\
  m &\triangleright m[\text{pc}\mapsto l']
\end{align*}
\]
Rules for executing commands

\[ m(pc) = l \quad P(l) = \text{IfZ } x \text{ Goto } l' \quad m(x) = 0 \]
\[ m \triangleright m[pc \mapsto l'] \]

\[ m(pc) = l \quad P(l) = \text{IfZ } x \text{ Goto } l' \quad m(x) \neq 0 \]
\[ m \triangleright m[pc \mapsto l+1] \]

\[ m(pc) = l \quad P(l) = \text{IfNZ } x \text{ Goto } l' \quad m(x) \neq 0 \]
\[ m \triangleright m[pc \mapsto l'] \]

\[ m(pc) = l \quad P(l) = \text{IfNZ } x \text{ Goto } l' \quad m(x) = 0 \]
\[ m \triangleright m[pc \mapsto l+1] \]
Executing programs

• For a program $P=1:c_1,...,n:c_n$ we define executions as finite or infinite sequences $m_1 \triangleright m_2 \triangleright ... \triangleright m_k \triangleright ...$

• We write $m \triangleright^* m'$ if there is a finite execution starting at $m$ and ending at $m'$:
  $$m = m_1 \triangleright m_2 \triangleright ... \triangleright m_k = m'$$
Semantics of a program

- For a program $P=1:c_1,\ldots,n:c_n$ and a state $m$ s.t. $m(pc)=1$ we define the result of executing $P$ on $m$ as

  $$\llbracket P \rrbracket m = \begin{cases} m' & \text{if } m \triangleright^* m' \text{ and } m'(pc)=n+1 \\ \bot & \text{else} \end{cases}$$

- **Lemma**: the function is well-defined (i.e., at most one output state)
Execution example

• Execute the following intermediate language program on a state where all variables evaluate to 0

```
1: t1 := 137
2: y := t1 + 3
3: IfZ x Goto 7
4: t2 := y
5: z := t2
6: Goto 9
7: t3 := y
8: x := t3
9: skip
```

$$m = [pc\rightarrow 1, \ t1\rightarrow 0, \ t2\rightarrow 0, \ t3\rightarrow 0, \ x\rightarrow 0, \ y\rightarrow 0] \triangleright ?$$
Execution example

• Execute the following intermediate language program on a state where all variables evaluate to 0

```
1: t1 := 137
2: y := t1 + 3
3: IfZ x Goto 7
4: t2 := y
5: z := t2
6: Goto 9
7: t3 := y
8: x := t3
9: skip
```

\[ m = [pc\rightarrow 1, \ t1\rightarrow 0, \ t2\rightarrow 0, \ t3\rightarrow 0, \ x\rightarrow 0, \ y\rightarrow 0] \uparrow \]
\[ m[pc\rightarrow 2, \ t1\rightarrow 137] \uparrow \]
\[ m[pc\rightarrow 3, \ t1\rightarrow 137, \ y\rightarrow 140] \uparrow \]
\[ m[pc\rightarrow 7, \ t1\rightarrow 137, \ y\rightarrow 140] \uparrow \]
\[ m[pc\rightarrow 8, \ t1\rightarrow 137, \ t3\rightarrow 140, \ y\rightarrow 140] \uparrow \]
\[ m[pc\rightarrow 9, \ t1\rightarrow 137, \ t3\rightarrow 140, \ , \ x\rightarrow 140, \ y\rightarrow 140] \uparrow \]
\[ m[pc\rightarrow 10, \ t1\rightarrow 137, \ t3\rightarrow 140, \ , \ x\rightarrow 140, \ y\rightarrow 140] \]
Semantic equivalence

• $P_1$ and $P_2$ are semantically equivalent if for all $m$ the following holds:

?
Semantic equivalence

- $P_1$ and $P_2$ are semantically equivalent if for all $m$ the following holds:

\[
\text{Attempt 1: } \begin{array}{c}
\llbracket P_1 \rrbracket m = \llbracket P_2 \rrbracket m
\end{array}
\]

Are these programs semantically equivalent?

```
1: t1 := 137
2: y := t1 + 3
```

```
1: y := 140
```
Semantic equivalence

- $P_1$ and $P_2$ are semantically equivalent if for all $m$ the following holds:

$$\left(\llbracket P_1 \rrbracket m \right)|_{\text{Var}} = \left(\llbracket P_2 \rrbracket m \right)|_{\text{Var}}$$
Exercise

Why do we need this?
ADDING PROCEDURES
**While+procedures syntax**

\[
\begin{align*}
A & \rightarrow n \mid x \mid A \text{ ArithOp } A \mid (A) \mid f(A^*) \\
\text{ArithOp} & \rightarrow - \mid + \mid * \mid / \\
B & \rightarrow \text{true} \mid \text{false} \\
& \quad \mid A = A \mid A \leq A \mid \neg B \mid B \land B \mid (B) \\
S & \rightarrow x := A \mid \text{skip} \mid S; S \mid \{ S \} \\
& \quad \mid \text{if } B \text{ then } S \text{ else } S \\
& \quad \mid \text{while } B \text{ } S \\
& \quad \mid f(A^*) \\
F & \rightarrow f(x^*) \{ S \} \\
P & ::= F^+
\end{align*}
\]

- \( n \in \text{Num} \) Numerals
- \( x \in \text{Var} \) Program variables
- \( f \) Function identifiers, including `main`

Can be used to call side-effecting functions and to return values from functions.
IL+procedures syntax

\[
\begin{align*}
V &\rightarrow n \mid x \\
R &\rightarrow V \text{ } Op \text{ } V \mid V \mid \text{Call } f(V^*) \mid p_n \\
Op &\rightarrow - \mid + \mid \ast \mid / \mid = \mid \leq \mid << \mid >> \mid \ldots \\
C &\rightarrow l: \text{skip} \\
&\mid l: x := R \\
&\mid l: \text{Goto } l' \\
&\mid l: \text{IfZ } x \text{Goto } l' \\
&\mid l: \text{IfNZ } x \text{Goto } l' \\
&\mid l: \text{Call } f(x^*) \\
&\mid l: \text{Ret } x \\
IR &\rightarrow C^+
\end{align*}
\]

- \(n \in \text{Num}\) Numerals
- \(l \in \text{Num}\) Labels and function identifiers
- \(x \in \text{Var} \cup \text{Temp} \cup \text{Params}\) Variables, temporaries, and parameters
- \(f\) Function identifiers

Accesses value of function parameter \(n\)
Exercise

Let the set of variables be $\text{Temp} \cup \text{Var}$.


FORMALIZING THE CORRECTNESS OF LOWERING
Exercise 1

• Are the following equivalent in your opinion?

**WHILE**

x := 137

**IL**

1: t1 := 137
2: x := t1
Exercise 2

• Are the following equivalent in your opinion?

```plaintext
WHILE

x := 137

IL

1: y := 137
2: x := y
```
Exercise 3

• Are the following equivalent in your opinion?

**WHILE**

\[
x := 137
\]

**IL**

2: \( t2 := 138 \)
3: \( t1 := 137 \)
4: \( x := t1 \)
Exercise 4

• Are the following equivalent in your opinion?

**WHILE**

x := 137

**IL**

2: t2 := 138
3: t1 := 137
4: x := t1
5: t1 := 138
Equivalence of arithmetic expressions

• While state: $\sigma \in \text{Var} \rightarrow \mathbb{Z}$
• IL state: $m \in (\text{Var} \cup \text{Temp} \cup \{pc\}) \rightarrow \mathbb{Z}$
• Define label($P$) to be first label of IL program $P$
• An arithmetic While expression $a$ is equivalent to an IL program $P$ and $x \in \text{Var} \cup \text{Temp}$ iff for every input state $m$ such that $m(\text{pc})=\text{label}(P)$:
  
  $\llbracket a \rrbracket m|_{\text{Var}} = (\llbracket P \rrbracket m) x$
Expression equivalence example

• $\llbracket 137 + x \rrbracket m \mid_{\text{Var}} = (\llbracket P \rrbracket m) \uparrow 3$
  for every state $m$ such that $m(pc)=1$

• Example: $m=\llbracket pc \mapsto 1 \, , t1 \mapsto 0 \, , t2 \mapsto 0 \, , t3 \mapsto 0 \, , x \mapsto 3 \rrbracket$
  then $m \mid_{\text{Var}}$ is $m \mid_{\{x\}} = \llbracket x \mapsto 3 \rrbracket$

\begin{align*}
\text{WHILE} & \quad \text{IL} \\
137 + x & \quad \begin{align*}
1: \ t1 & := x \\
2: \ t2 & := 137 \\
3: \ t3 & := t1 + t2
\end{align*}
\end{align*}
Statement equivalence

• We define state equivalence by
  \[ \sigma \approx m \iff \sigma = m \mid_{\text{Var}} \]
  – That is, for each \( x \in \text{Var} \): \( \sigma(x) = m(x) \)

• A While statement \( S \) is equivalent to an IL program \( P \)
  iff for every input state \( m \) such that \( m(pc) = \text{label}(P) \):
  \[ \llbracket S \rrbracket_{\text{Var}} m \approx \llbracket P \rrbracket m \]
Example

- $\llbracket S \rrbracket m \mid_{\text{Var}} \approx \llbracket P \rrbracket m$
- Example: $m = [\text{pc} \mapsto 1, \text{t1} \mapsto -1, \text{x} \mapsto 3]$ then $m \mid_{\text{Var}}$ is $m \mid_{\{x\}} = [x \mapsto 3]$
- $\llbracket S \rrbracket m \mid_{\text{Var}} = [x \mapsto 137]$
- $\llbracket P \rrbracket m = [\text{pc} \mapsto 3, \text{t1} \mapsto 137, \text{x} \mapsto 137]$

**WHILE**

1. $\text{t1} := 137$
2. $\text{x} := \text{t1}$

**IL**
Next lecture: Dataflow-based Optimizations