### Example: Increment

First, let's add functions to our language.

```latex
\text{Example: Increment}
```

For example, a function that increments its input:

```latex
\text{Example: Increment}
```

```latex
\text{Let t0 = 1, t1 = t0 + 1, t2 = t1 + 1, t3 = t2 + 1, t4 = t3 + 1. Then, t4 = 5.}
```

As always, let's look at some examples.

#### Functions

We will use a function to extract the bindings.

```latex
\text{Function application}
```

```latex
\text{Let t = f(x), then we can write t = f(x)}
```

### Example: Mutually Recursive Functions

Finally, let's add some new types to represent programs.

```latex
\text{Example: Mutually Recursive Functions}
```

For this language, any function can call any other function.

### Example: Factorial

Let's create a special type that represents places where data is bound.

```latex
\text{Example: Factorial}
```

```latex
\text{Let n be a positive integer, then \text{factorial}(n) returns the product of all positive integers up to n.}
```

```latex
\text{For example, \text{factorial}(5) = 120.}
```

```latex
\text{This program should produce the result 120.}
```

```latex
\text{We have a function definition followed by a single "main" expression, which is evaluated to yield the program's result.}
```

```latex
\text{For example, a function that increments its input:}
```

```latex
\text{Example: Increment}
```

```latex
\text{Let t0 = 1, t1 = t0 + 1, t2 = t1 + 1, t3 = t2 + 1, t4 = t3 + 1. Then, t4 = 5.}
```

```latex
\text{As always, let's look at some examples.}
```

### Types

Let's re-examine some properties of the program:

```latex
\text{Example: Types}
```

```latex
\text{The syntactic type of the program does not denote a function type.}
```

```latex
\text{This lets us write programs that don't have a "main" expression, and which can be evaluated to yield the program's result.}
```

```latex
\text{For example, a function that increments its input:}
```

```latex
\text{Example: Increment}
```

```latex
\text{Let t0 = 1, t1 = t0 + 1, t2 = t1 + 1, t3 = t2 + 1, t4 = t3 + 1. Then, t4 = 5.}
```

```latex
\text{As always, let's look at some examples.}
```

### Functions

We will use a function to extract the bindings.

```latex
\text{Function application}
```

```latex
\text{Let t = f(x), then we can write t = f(x)}
```

### Example: Mutually Recursive Functions

Finally, let's add some new types to represent programs.

```latex
\text{Example: Mutually Recursive Functions}
```

For this language, any function can call any other function.

### Example: Factorial

Let's create a special type that represents places where data is bound.

```latex
\text{Example: Factorial}
```

```latex
\text{Let n be a positive integer, then \text{factorial}(n) returns the product of all positive integers up to n.}
```

```latex
\text{For example, \text{factorial}(5) = 120.}
```

```latex
\text{This program should produce the result 120.}
```

```latex
\text{We have a function definition followed by a single "main" expression, which is evaluated to yield the program's result.}
```

```latex
\text{For example, a function that increments its input:}
```

```latex
\text{Example: Increment}
```

```latex
\text{Let t0 = 1, t1 = t0 + 1, t2 = t1 + 1, t3 = t2 + 1, t4 = t3 + 1. Then, t4 = 5.}
```

```latex
\text{As always, let's look at some examples.}
```

### Types

Let's re-examine some properties of the program:

```latex
\text{Example: Types}
```

```latex
\text{The syntactic type of the program does not denote a function type.}
```

```latex
\text{This lets us write programs that don't have a "main" expression, and which can be evaluated to yield the program's result.}
```

```latex
\text{For example, a function that increments its input:}
```

```latex
\text{Example: Increment}
```

```latex
\text{Let t0 = 1, t1 = t0 + 1, t2 = t1 + 1, t3 = t2 + 1, t4 = t3 + 1. Then, t4 = 5.}
```

```latex
\text{As always, let's look at some examples.}
```

### Functions

We will use a function to extract the bindings.

```latex
\text{Function application}
```

```latex
\text{Let t = f(x), then we can write t = f(x)}
```

### Example: Mutually Recursive Functions

Finally, let's add some new types to represent programs.

```latex
\text{Example: Mutually Recursive Functions}
```

For this language, any function can call any other function.

### Example: Factorial

Let's create a special type that represents places where data is bound.

```latex
\text{Example: Factorial}
```

```latex
\text{Let n be a positive integer, then \text{factorial}(n) returns the product of all positive integers up to n.}
```

```latex
\text{For example, \text{factorial}(5) = 120.}
```

```latex
\text{This program should produce the result 120.}
```

```latex
\text{We have a function definition followed by a single "main" expression, which is evaluated to yield the program's result.}
```

```latex
\text{For example, a function that increments its input:}
```

```latex
\text{Example: Increment}
```

```latex
\text{Let t0 = 1, t1 = t0 + 1, t2 = t1 + 1, t3 = t2 + 1, t4 = t3 + 1. Then, t4 = 5.}
```

```latex
\text{As always, let's look at some examples.}
```

### Types

Let's re-examine some properties of the program:

```latex
\text{Example: Types}
```

```latex
\text{The syntactic type of the program does not denote a function type.}
```

```latex
\text{This lets us write programs that don't have a "main" expression, and which can be evaluated to yield the program's result.}
```

```latex
\text{For example, a function that increments its input:}
```

```latex
\text{Example: Increment}
```
Which function(s) would we have to modify to add QUIZ

The bulk of the work is done by three functions

Thus, our return the which will possible

To make using a language and compiler pleasant, lets return

Its rather irritating to get errors one-by-one.

Catching Multiple Errors

In the above, we have defined the types:

Next, lets insert a

Transforms

the result.

We can

We make it an

Types: An Error Reporting API

Next, we will look at an

2. Static Checking

Increasingly,

There is a huge spectrum of checks possible:

bugs in the code at compile time

Next, we will look at an

4. Code Linting

2. Static Well-formedness Checking

We get

We would like compilation to fail, not silently, but with useful messages:

Static Typing

Dependent or

Contract Checking

Modern compiler.

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler

is the most important phase of a compiler, and modern compiler
we must Thus to compile each definition

We are using the

We will use our old friend,

Now, lets look at

3. Compiling Functions

and traverses the expression:

The helper function

Let's look at how we might check for two types of errors:

QUIZ

Which function(s) would we have to modify to add

QUIZ

body
Tail Recursion

But, the "equivalent" overhead). For example (the expensive loops). Our language doesn't have
tail recursive loops. As an example, of its behavior, consider the (source) program:

```
def add2(x, y):
    return x + y
```

The code for the above call is generated by itself. As an example, of its behavior, consider the (source) program:

```
def sumTo(n):
    return 
```

To compile a single

Implementation

To compile a single

### Compiling Programs

1. **Create a block**
   - Label for start of function
   - Label the 'body' for tail-calls
   - Push parameters onto the stack
   - Copy parameters into stack slots
   - Copy remaining stack args
   - Copy upto 6 register args
   - Copy from RBP-offset-src to RBP-offset-dst
   - Execute 'body' with result in RAX

1. **Compile the whole program**
   - Allocate n local-vars
   - Save caller's RBP
   - Pop the stack by incrementing
   - Return to caller

Data

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Representation of numbers</td>
</tr>
<tr>
<td>String</td>
<td>Representation of text sequences</td>
</tr>
<tr>
<td>Boolean</td>
<td>Representation of truth values</td>
</tr>
<tr>
<td>Float</td>
<td>Representation of real numbers</td>
</tr>
<tr>
<td>Symbol</td>
<td>Representation of identifiers</td>
</tr>
<tr>
<td>List</td>
<td>Representation of ordered sequences</td>
</tr>
<tr>
<td>Map</td>
<td>Representation of key-value pairs</td>
</tr>
<tr>
<td>Set</td>
<td>Representation of unordered sets</td>
</tr>
<tr>
<td>Record</td>
<td>Representation of named tuples</td>
</tr>
<tr>
<td>Function</td>
<td>Representation of first-class objects</td>
</tr>
</tbody>
</table>

Programs

```
add2(12, 7)
```

Can be implemented with 2 registers

```
fac(9999)
```

Requires a

```
fac(10000)
```

Uses

```
fac(10000)
```

Uses

```
fac(10000)
```

Uses

```
fac(10000)
```

Uses

```
fac(10000)
```

Uses

```
fac(10000)
```

Uses

```
sumTo(n)
```

Can be implemented with 2 registers

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```

Uses

```
sumTo(12)
```
## Compilation of Tail Calls

### Exercise

#### Algorithm:

1. Identify the tail calls following the above strategy.
2. Implement the tail calls in place, i.e., we could identify the tail calls during
   the above strategy, we need a way to:

#### Requirements:

- The results are each call added to the parameter at that frame.
- 
  
- No need to use call-stack, can make recursive call

#### Tail Recursive Strategy

Here's the code for compileExpr env (s" are non-tail

### Examples

#### Defining the Types:

```plaintext
Labeling
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Source Span</th>
<th>Expression has unique tag</th>
<th>Expression has source position metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SourceSpan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prim2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Defining the Functions:

```plaintext
Jump
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Source Span</th>
<th>Expression has unique tag</th>
<th>Expression has source position metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Executing the Program:

```plaintext
Tailcall
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Source Span</th>
<th>Expression has unique tag</th>
<th>Expression has source position metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prim2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Executing the Program

```plaintext
Jump
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Source Span</th>
<th>Expression has unique tag</th>
<th>Expression has source position metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Executing the Program

```plaintext
Jump
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Source Span</th>
<th>Expression has unique tag</th>
<th>Expression has source position metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```