The λ-Calculus
Programming Languages (CSCI 3300)

Prof. Harley Eades (heades@gru.edu).

In this lecture we are going to start to study our first functional programming languages called the λ-Calculus. This language was first proposed by Alonzo Church in the early 1900's. He was studying the foundations of mathematics and thought it might be useful for studying logic. Later, he realized that it is a model of computation. That is, it is just as powerful as Turing Machines which implies it is just as powerful as modern day computers. In fact, Turing Machines and the λ-calculus are equivalent. This is known as the Church-Turing thesis. The thing that sets the λ-calculus apart is that it is extermely simple.

Today the λ-calculus is as important as ever, and can be found at the heart of all functional programming languages. This includes Haskell. So when one is programming in Haskell they are programming in the λ-calculus.

1 The Language of the λ-Calculus

The λ-calculus is extermely simple. In fact, it essentially just consists of functions and function application. The following defines the grammar for the λ-calculus:

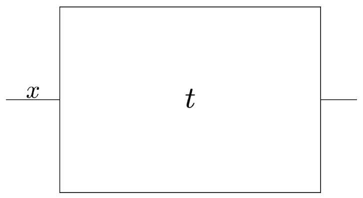
$$\begin{array}{rl} \text{(Variables)} & v ::= x \mid y \mid z \mid \ldots \\ \text{(Terms)} & t ::= v \mid \lambda x.t \mid t_1\ t_2 \end{array}$$

The language of the λ-calculus consists of an infinite number of variables, anonymous unary functions, and function application. That's all.

Variables are denoted x, y, and z. We also subscript variables to help distinguish them. Keep in mind that there are an infinite number of variable names, and we can name a variable anything we wish. However, it is common to stick to single letter names with subscripts.

Variables are not the samething as variables in object oriented programming languages or imparative languages like C, C++, C#, and Java. Variables in programming languages like those are pointers to memeory, and can be reassigned multiple times. However, in the λ-calculus there is no memory. One should think of the variables in object oriented and imparative programming languages as "assignables" instead of variables, and think of the variables of the λ-calculus as the variables one finds in mathematics. The variables in mathematics and the λ-calculus are **replaced** with other objects and not assigned. We will see more of this replacment notion below.

Arguably the most important part of the λ-calculus is the λ-abstraction (or anonymous unary function) denoted $\lambda x.t$. We all x the bound variable, and t the body of the λ-expression. Here the variable x is the placeholder of the input to the function. Keep in mind that the argument variable – here x – can be any variable at all. It does not have to always be called x. The best way to think of these functions are as black boxes with only one input and one output:



The final piece of the language of the λ-calculus is function application denoted $t_1\, t_2$. We will see that this is the locus of computation. Programs will be defined as functions, and running programs will consist of simplifying function applications. How this simplifcation works will soon be discussed. A note on terminology. It is common to refer to terms as expressions and programs in general. I will definately use this terms interchangeable.

2 Example λ-Calculus Programs

Here we give several example programs:

- The identity function: $\lambda x.x$.
- Forget the first $\lambda x.\lambda y.x$.
- Forget the second $\lambda x.\lambda y.y$.
- The successor function $\lambda s.\lambda x.s\, z$.
- Application: $(\lambda x.x)\, y$.
- Pairs: $\lambda x.\lambda y.\lambda z.(z\, x)\, y$.
- The first projection function: $\lambda p.p\,(\lambda x.\lambda y.x)$.
- The second projection function: $\lambda p.p\,(\lambda x.\lambda y.y)$.

3 Free and Bound Variables

One of the more confusing notions to understand when it comes to variables in the λ-calculus is their dual role. In one respect they can be seen as a global variable, while in another they are to be considered as local variables.

We call global variables "free variables," and when we write something like y we call y a free variable. In addition, the variables in $(x\, y)\, z$ are all free variables. Similarly, the variable x in $\lambda y.x$ is a free variable, but what about y?

A variable whose name is the input to a function is called a "bound variable." For example, the variable x is bound in $\lambda x.x$. A bound variable is a locally scoped one. It is only useable from within the body of its λ-abstraction. Every variable in $\lambda x.\lambda y.x\, y$ is bound. How about the rightmost x in $(\lambda x.z)\, x$? It is in fact, not bound, but free. Note that for some λ-abstraction $\lambda x.t$ we say that x is bound in t, and we call the λx part the binder of the λ-abstraction. So in the previous example $(\lambda x.z)\, x$ we say that x is bound in z, but the rightmost x is free in the entire term.

There is a pretty profound thing that occurs when we take a λ-abstraction, say $\lambda x.x\, y$, and remove its binder to obtain $x\, y$. The variable x changes from a bound variable to a free variable. That is, it changes from a locally scoped variable to a globally scoped variable. This may seem like a small point, but when we start implementing programming languages with bound variables this has profound implications. It becomes not so easy to implement. We will see more of this later in the semester.

4 Parsing Conventions and Scope

We will be dealing with a lot of λ-expressions. So it convenient to have parsing conventions to make writing down λ-expressions eaiser.

First, λ-abstractions consume everything to the left of the period, and this range also happens to be the scope of the input variable. So for example, consider the term $\lambda x.\lambda y.x\;z\;s\;s'$. The parsing convention is that the body of the outter-most λ-abstraction is $\lambda y.x\;z\;s\;s'$. That is it extends all the way to the right most side of the expression. This also happens to be the scope of the input variable x. So in $(\lambda x.x\;y)\;x$ the rightmost x is not associated with the bound variable x, in fact, these are two completely different variables – the first is bound while the second is free. To prevent ambiguity with scope use parentheses to delimit the bounds. Reconsidering this term $\lambda x.\lambda y.x\;z\;s\;s'$ we can add parentheses to prevent ambiguity, that is, $\lambda x.(\lambda y.(x\;z\;s\;s'))$ fully delimits the scope of each λ-abstraction.

There is one other parsing convention that we will use. Function application associates to the left. That is, the term $t_1\;t_2\;t_3\;t_4\;t_5$ is equivalent to $(((t_1\;t_2)\;t_3)\;t_4)\;t_5$.

Now using these conventions we can fully parenthesize λ-expressions to prevent ambiguity. So the fully parenthesized version of $\lambda x.\lambda y.\lambda z.y\;z\;x\;w$ is $\lambda x.(\lambda y.(\lambda z.(((y\;z)\;x)\;w)))$.

5 Structural Recursion

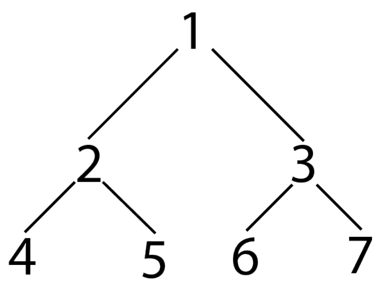
At this point we have only seen the language of the λ-calculus, but this is supposed to be a programming language, so how do we compute with these things? Before we can answer that we have to understand how we can define functions on the language of the λ-calculus, and in order to do this we must understand structural recursion.

Lets consider an example first. Suppose we wanted to define a function that operates on λ-calculus expressions (or simply λ-expressions) that when given a λ-expression computes its depth when considered as a tree. How would we do this? The following accompishes this:

$$\begin{aligned}
\mathsf{depth}(x) &= 1\\
\mathsf{depth}(\lambda x.t) &= 1 + \mathsf{depth}(t)\\
\mathsf{depth}(t_1\;t_2) &= \mathsf{depth}(t_1) + \mathsf{depth}(t_2)
\end{aligned}$$

The depth function takes a λ-expression as an arugment and produces a number greater than or equal to 1. This function is defined using the notion of **structural recursion**.

Structural recursion is a device for defining functions over data that is built up from smaller pieces of data. For example, trees are built up from smaller pieces of data. We know a binary tree consists of a leaf and a node that consists of two branches. If we denote a leaf by $\mathsf{leaf}\,n$ where n can be any natural number, and we denote nodes as $\mathsf{node}\,n\,T_1\,T_2$ where T_1 and T_2 are the two child trees, then we can define any binary tree at all. An example would be $\mathsf{node}\,1\,(\mathsf{node}\,2\,(\mathsf{leaf}\,4)\,(\mathsf{leaf}\,5))\,(\mathsf{node}\,3\,(\mathsf{leaf}\,6)\,(\mathsf{leaf}\,7)))$. This example in graphical form is:



Now trees have a clearly defined structure. A leaf has no constituent tree-parts, but a node has two constituent tree-parts: the left subtree and the right subtree. So using this idea we can define an ordering on trees:

$$\begin{array}{l} \mathsf{node}\, n\, T_1\, T_2 > T_1 \\ \mathsf{node}\, n\, T_1\, T_2 > T_2 \end{array}$$

Notice that in the above definition we do not mention leafs and that is because there is nothing smaller than a leaf. The previous ordering is the subtree ordering on binary trees.

The idea that we are trying to get accross here is that any datatype which can be built up from smaller pieces contains a natural substructure ordering. This can then be used to define functions by recursion. Remember, a recursive function has at least one base case, and at least one step case. For the case of trees, the base case would be for leafs, and the step case would be for nodes. Similarly, for the case of λ-expressions the base case is for variables, and the step case is for λ-abstractions and application. Then we are allowed to make recursive calls on the subdata whether it be a subtree or a sub-λ-expression.

Lets consider an example function defined by structural recursion on trees. The following computes the set of tree data given any tree:

$$\begin{array}{rl} \mathsf{flatten}(\mathsf{leaf}\, n) & = \{n\} \\ \mathsf{flatten}(\mathsf{node}\, m\, T_1\, T_2) & = \{m\} \cup \mathsf{flatten}(T_1) \cup \mathsf{flatten}(T_2) \end{array}$$

Now flattening our example tree from above we obtain:

$$\begin{array}{l} \mathsf{flatten}(\mathsf{node}\, 1\, (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7))) \\ \quad = \{1\} \cup \mathsf{flatten}(\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5)) \cup \mathsf{flatten}(\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7)) \\ \quad = \{1\} \cup \{2\} \cup \mathsf{flatten}(\mathsf{leaf}\, 4) \cup \mathsf{flatten}(\mathsf{leaf}\, 5) \cup \mathsf{flatten}(\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7)) \\ \quad = \{1\} \cup \{2\} \cup \mathsf{flatten}(\mathsf{leaf}\, 4) \cup \mathsf{flatten}(\mathsf{leaf}\, 5) \cup \{3\} \cup \mathsf{flatten}(\mathsf{leaf}\, 6) \cup \mathsf{flatten}(\mathsf{leaf}\, 7) \\ \quad = \{1\} \cup \{2\} \cup \{4\} \cup \mathsf{flatten}(\mathsf{leaf}\, 5) \cup \{3\} \cup \mathsf{flatten}(\mathsf{leaf}\, 6) \cup \mathsf{flatten}(\mathsf{leaf}\, 7) \\ \quad = \{1\} \cup \{2\} \cup \{4\} \cup \{5\} \cup \{3\} \cup \mathsf{flatten}(\mathsf{leaf}\, 6) \cup \mathsf{flatten}(\mathsf{leaf}\, 7) \\ \quad = \{1\} \cup \{2\} \cup \{4\} \cup \{5\} \cup \{3\} \cup \{6\} \cup \mathsf{flatten}(\mathsf{leaf}\, 7) \\ \quad = \{1\} \cup \{2\} \cup \{4\} \cup \{5\} \cup \{3\} \cup \{6\} \cup \{7\} \\ \quad = \{1, 2, 4, 5, 3, 6, 7\} \end{array}$$

Getting back to the λ-calculus we can now use structural recursion to define functions over λ-expressions just like the depth function from above. In fact, just as we saw for trees there is a natural structural ordering on λ-expressions:

$$\begin{array}{l} \lambda x.t > t \\ t_1\, t_2 > t_1 \\ t_1\, t_2 > t_2 \end{array}$$

Again, variables are the base case, so they do not appear in the above definition. Now notice that the above ordering is terminating, that is, given an arbitrary term, it can only get smaller for a finite amount of time. That is, eventually we must bottom out at a variable. The fact that this ordering must terminate impilies that the depth function from above must always produce an output.

Functions defined on languages like the λ-expressions are known as metafunctions, because they can be thought of as living outside the actually language. We will see a number of these throughout the semester.

6 Variable Substitution

I mentioned above that a variables job is to be replaced by other λ-expressions. At this point we define how this replacement is done, and then use this new definition to define how to compute with λ-expressions.

The substitution meta-function on λ-expressions is defined as follows:

$$[t/x]x = t$$

$$[t/x]y = y \quad \text{where } y \text{ is distinct from } x$$

$$[t/x](\lambda y.t') = \lambda y.[t/x]t' \quad \text{where } y \text{ is distinct from } x$$

$$[t/x](\lambda x.t') = \lambda x.t'$$

$$[t/x](t_1\, t_2) = ([t/x]t_1)\,([t/x]t_2)$$

The substitution function denoted $[t/x]t'$ takes in three arguments: t, x, and t'. We read it as "substitute t for x in t'." Intuitively, we can think of it as simply replacing every occurrence of x with t in t'. However, one has to be careful to not replace a bound variable. The second and third conditions in the above definition prevent this from happening. In Section 3 we discussed the difference between bound variables and free variables. **We are only allowed to replace free variables using substitution.**

Some examples:

- $[y/x]x = y$
- $[y/x]z = z$ where z is distinct from x
- $[y/z](\lambda x.z) = \lambda x.[y/z]z = \lambda x.y$
- $[y/z]((\lambda x.x)\, z) = ([y/z](\lambda x.x))\,([y/z]z) = (\lambda x.[y/z]x)\,([y/z]z) = (\lambda x.x)\, y$
- $[\lambda x.\lambda y.y/z]((z\, s)\, s') = ([\lambda x.\lambda y.y/z](z\, s))\,([\lambda x.\lambda y.y/z]s')$
  $= (([\lambda x.\lambda y.y/z]z)\,([\lambda x.\lambda y.y/z]s))\,([\lambda x.\lambda y.y/z]s')$
  $= ((\lambda x.\lambda y.y)\, s)\, s'$

7 Inference Systems

A lot of the design of programming languages in general involve the careful definition of certain algorithms that manipulate programs. This is similar to the idea of defining functions on programs like the `depth` function we saw in class. A large number of algorithms can be described using inference systems.

An inference system consists of one or more inference rules. An inference rule has the following shape:

$$\frac{P_1 \qquad \cdots \qquad P_n}{C}\ \textsc{Name}$$

We call $P_1, \cdots, P_n$ the premises of the rule, and C the conclusion. Note that it is completely satisfactory to have an inference rule with no premises at all. In fact, an inference rules with no premises are called axioms. Inference rules are read from the top down as an if-statement. So the above rule should be read "If P_1, P_2, $\ldots$, and P_n all hold, then C holds." If an inference rule has no premises at all, then its conclusion always holds, hence, the term axiom.

An example may help. The following set of inference rules define what it means to be a binary tree:

$$\dfrac{n \in \mathbb{N}}{(\mathsf{leaf}\, n)\, Tree}\ \textsc{leaf} \qquad \dfrac{T_1\, Tree \qquad T_2\, Tree \qquad n \in \mathbb{N}}{(\mathsf{node}\, n\, T_1\, T_2)\, Tree}\ \textsc{node}$$

The first rule states, "for all $n \in \mathbb{N}$, $\mathsf{leaf}\, n$ is a tree." Now the second states that "for any T_1, T_2, and $n \in \mathbb{N}$, if T_1 and T_2 are trees, then $\mathsf{node}\, n\, T_1\, T_2$ is a tree."

So what are these rules good for? Inference system should be thought of as an algorithm where the conclusion, C, is an input, and the output essentially amounts to yes or no. The former indicating that the input adheres to the rules, and no otherwise. So for example, given the rules above we can ask is $\mathsf{node}\, 1\, (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7))$ a tree, but how can we actually answer this question? We answer this question by building derivations of the rules. This is all very similar to how we used grammars.

A derivation is the application of one or more rules. For example, one of the simplest derivations using the rules for trees is the following:

$$\dfrac{1 \in \mathbb{N}}{(\mathsf{leaf}\, 1)\, Tree}\ \textsc{leaf}$$

In the above $\mathsf{leaf}\, 1$ is the input, and the derivation consists of only the rule `leaf`, and this successfully shows that $\mathsf{leaf}\, 1$ is indeed a tree. A more complete derivation is the follows:

$$\dfrac{\dfrac{\dfrac{4 \in \mathbb{N}}{(\mathsf{leaf}\, 4)\, Tree}\ \textsc{leaf} \quad \dfrac{5 \in \mathbb{N}}{(\mathsf{leaf}\, 5)\, Tree}\ \textsc{leaf} \quad 2 \in \mathbb{N}}{(\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, Tree}\ \textsc{node} \quad \dfrac{\dfrac{6 \in \mathbb{N}}{(\mathsf{leaf}\, 6)\, Tree}\ \textsc{leaf} \quad \dfrac{7 \in \mathbb{N}}{(\mathsf{leaf}\, 7)\, Tree}\ \textsc{leaf} \quad 3 \in \mathbb{N}}{(\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7))\, Tree}\ \textsc{node} \quad 1 \in \mathbb{N}}{(\mathsf{node}\, 1\, (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7)))\, Tree}\ \textsc{node}$$

Derivations are constructed from the bottom up. In the above we begin by asking the question, "does $(\mathsf{node}\, 1\, (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7)))\, Tree$ hold?" To prove that it holds we try to apply the rules of the tree inference system one at a time, from the bottom up. So after asking the question we try and pattern match on the conclusion of each rule. If what we are asking matches the conclusion of the rule, then we may then move to the premises of that rule. For example, we construct the above derivation by starting like this, we first asking the question:

$$(\mathsf{node}\, 1\, (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7)))\, Tree$$

Then we try to pattern match on the previous statement against the conclusion of each rule, and we can see that it matches the conclusion of the `node` rule, where $n = 1$, $T_1 = (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))$, and $T_2 = (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7))$. So we are allowed to apply that rule:

$$\dfrac{(\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, Tree \qquad (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7))\, Tree \qquad 1 \in \mathbb{N}}{(\mathsf{node}\, 1\, (\mathsf{node}\, 2\, (\mathsf{leaf}\, 4)\, (\mathsf{leaf}\, 5))\, (\mathsf{node}\, 3\, (\mathsf{leaf}\, 6)\, (\mathsf{leaf}\, 7)))\, Tree}\ \textsc{node}$$

Then we ask, which rule can be applied to each premise? Then we proceed until no further rules can be applied. Eventually we end up with the derivation from above.

Now how do we know when a derivation is successful? First, notice that the derivations are themselves tree structures. The axioms are the leaves of the tree, and the premises start branches of the tree. For example, in the above example of the complete derivation using the tree inference system we can see that the `node` rule forms two branches, one for each subtree. Now a derivation is **successful** if and only if every branch of the derivation ends in an axiom. That is, the very last rules on the top of the derivation must be all axioms.

If not, then the derivation is a failure, and the conclusion is said not to hold. The following is an example of a failure:

$$\dfrac{\dfrac{2 \in \mathbb{N}}{(\mathsf{leaf}\,2)\,Tree}\,\text{LEAF} \qquad 42 \qquad 1 \in \mathbb{N}}{(\mathsf{node}\,1\,(\mathsf{leaf}\,2)\,42)\,Tree}\,\text{NODE}$$

The previous derivation fails, because there is no rule to apply 42 to and thus the derivation does not end in all axioms – that is the `leaf` rule. Therefore, we must conclude that $\mathsf{node}\,1\,(\mathsf{leaf}\,2)\,42$ is not a valid tree.

Inference systems usually have a particular goal called a judgment. The inference system is said to judge whether not something is provable or not. For example, the tree inference system judges whether or not some mathematical object is a tree, and we can read the judgement $t\,Tree$ as "t is a tree." We then say that a judgment holds or is provable or is derivable if and only if we can construct a successful derivation of it.

Inference systems are everywhere in programming language design. They allow for a very nice, compact, and easy definitions of algorithms on the objects of a programming language. Now we can use inference systems to define how we can compute in the λ-calculus.

8 Computing in the λ-Calculus

The locus of computation in the λ-calculus is function application, $t_1\,t_2$, but even more specifically it is the application of a λ-abstraction to an argument, that is, $(\lambda x.t)\,t'$. It turns out that computation in the λ-calculus is really just a matter of replacing every occurrence of x in t with t', but how do we do this formally? We use an inference system:

$$\dfrac{}{(\lambda x.t)\,t' \leadsto [t'/x]t}\;\textsc{Beta} \qquad \dfrac{t \leadsto t'}{\lambda x.t \leadsto \lambda x.t'}\;\textsc{Lam}$$

$$\dfrac{t_1 \leadsto t_1'}{t_1\,t_2 \leadsto t_1'\,t_2}\;\textsc{App1} \qquad \dfrac{t_2 \leadsto t_2'}{t_1\,t_2 \leadsto t_1\,t_2'}\;\textsc{App2}$$

The judgment here is $t_1 \leadsto t_2$ which can be read as t_1 reduces to t_2. We call this set of inference rules the reduction rules for the λ-calculus. Note that one should think of the terms in these rules as schemas of terms that we match pattern against. So some care has to be taken when checking to see if a rule applies. The reduction rules should be used just as we used the tree inference system above. That is, we begin with a judgment that looks like $t_1 \leadsto t_2$, and then try to prove that t_1 reduces to t_2 if we can construct a successful derivation of $t_1 \leadsto t_2$ using the reduction rules. We can think of this style as starting with some solution to a problem, and then using the rules to check to see if the solution is correct.

The most important rule is the β-rule:

$$\dfrac{}{(\lambda x.t)\,t' \leadsto [t'/x]t}\;\textsc{Beta}$$

This rule says that any term matching the pattern $(\lambda x.t)\,t'$ can be replaced by $[t'/x]t$ which replaces every occurrence of x with t' in t.

Definition 1 (Redex)**.** *Any term matching $(\lambda x.t)\,t'$ is called a **redex**.*

So we can see that the term in the lefthand side of the arrow in the β-rule is the redex, and we call the righthand side the **contractum**. Notice that variable substitution is an important part of the β-rule, and now we can see that computing really amounts to using the β-rule which amounts to just using the variable substitution function to simplify function applications.

Now lets consider an example using the β-rules. The following shows that $(\lambda x.x)\,y$ reduces to y.

$$\frac{}{(\lambda x.x)\,y \rightsquigarrow y}\ \textsc{Beta}$$

This holds because the term $(\lambda x.x)\,y$ matches the pattern $(\lambda x.t)\,t'$ where the binder x is x, t is x, and t' is t, and $[y/x]x = y$. So sometimes this rule is hard to apply because we have to reverse the substitution function, that is, we had to realize the y is the same as $[y/x]x$ when just given y.

The reduction rules are literal. One can only apply the rule if the terms given match the term patterns in the conclusion. For example, can we apply the β-rule to conclude $x\,((\lambda y.y\,z)\,r) \rightsquigarrow x\,(r\,z)$? No, because $x\,((\lambda y.y\,z)\,r)$ does not match the pattern $(\lambda x.t)\,t'$, and $x\,(r\,z)$ does not match the pattern $[t'/x]t$ so that rule does not apply, but we should be allowed to reduce that redex, right? This is the job for the other reduction rules.

There are three other reduction rules:

$$\frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}\ \textsc{Lam} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\,t_2 \rightsquigarrow t_1'\,t_2}\ \textsc{App1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{t_1\,t_2 \rightsquigarrow t_1\,t_2'}\ \textsc{App2}$$

These rules allow us to move inside of terms. For example, we can now show $x\,((\lambda y.y\,z)\,r) \rightsquigarrow x\,(r\,z)$:

$$\frac{\dfrac{}{(\lambda y.y\,z)\,r \rightsquigarrow (r\,z)}\ \textsc{Beta}}{x\,((\lambda y.y\,z)\,r) \rightsquigarrow x\,(r\,z)}\ \textsc{App2}$$

Consider this term for example, $(\lambda x.x\,x)\,((\lambda y.y)\,r)$, how many redexes are there in this term? There are two, the entier term, and the subterm $(\lambda y.y)\,r$. So how do we reduce them both? In fact, we cannot do this, because these rules only support reducing exactly one redex. In fact, the reduction rules we have considered so far are called the **single-step full β-reduction.** The "full" part has to do with the ability to contract a redex anywhere within a term.

To allow for the reduction of multiple redexes we define a new inference system called **multi-step full β-reduction.** The inference system is as follows:

$$\frac{}{t \rightsquigarrow^* t}\ \textsc{Refl} \qquad \frac{t \rightsquigarrow t'}{t \rightsquigarrow^* t'}\ \textsc{SStep} \qquad \frac{t_1 \rightsquigarrow^* t_2 \quad t_2 \rightsquigarrow^* t_3}{t_1 \rightsquigarrow^* t_3}\ \textsc{MStep}$$

Now we can use this inference system to reduce more than one redex at a time. For example we can prove the following reductions (full derivations in class):

$$x\,((\lambda y.y)\,((\lambda z.z\,z)\,r)) \rightsquigarrow^* x\,(r\,r)$$
$$\lambda y.(\lambda x.\lambda y.y)\,((\lambda x.x\,x)\,(\lambda x.x\,x))\,z \rightsquigarrow^* \lambda y.z$$

There is one downfall of applying the reduction rules the way that we have. The redex is often the program the programmer has written, and the contractum is either the solution, or an intermediate point to the solution. So from here on out we – unless directed differently like in homeworks or exams – we are going to adopted a different way of applying these rules. We will use these rules a guiding principle for how to transform a term into another by contracting its redexes. So instead of writing down long and big derivations we will simply write down a chain of transformations. These transformations a very similar to the derivations we did on grammars. Lets consider an example.

$$\begin{aligned}
&\underline{(\lambda n.\lambda m.(\lambda s.\lambda z.(n\,s)\,(m\,s\,z)))\,(\lambda s.\lambda z.s\,z)\,(\lambda s.\lambda z.s\,(s\,z))}\\
&\qquad \rightsquigarrow \underline{(\lambda m.(\lambda s.\lambda z.((\lambda s.\lambda z.s\,z\,s)\,(m\,s\,z))))\,(\lambda s.\lambda z.s\,(s\,z))}\\
&\qquad \rightsquigarrow \lambda s.\lambda z.(((\lambda s.\lambda z.s\,z)\,s)\,\underline{((\lambda s.\lambda z.s\,(s\,z))\,s\,z})\\
&\qquad \rightsquigarrow \lambda s.\lambda z.(\underline{((\lambda s.\lambda z.s\,z)\,s)}\,(s\,(s\,z)))\\
&\qquad \rightsquigarrow \lambda s.\lambda z.\underline{(\lambda z.s\,z)\,(s\,(s\,z))}\\
&\qquad \rightsquigarrow \lambda s.\lambda z.s\,(s\,(s\,z))
\end{aligned}$$

Notice how in the above I underlined the redex I was contracting in each step. This is required when writing these down in your homework and exams, because it makes it apparent which redex we are contracting at each step. Also, notice that where we can put the underline is determined by the rule of the inference system. Lastly, each transformation is step in the single-step full β-reduction inference system. However, if one can give a chain of one or more transformations from t_1 to t_2, then we can derive $t_1 \rightsquigarrow^* t_2$ in the multi-step full β-reduction inference system.