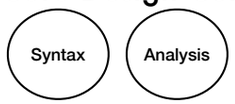
Core Design Concepts Discussed:



Inductive Definitions

Inference Rules, Deductions, and Logic Programming

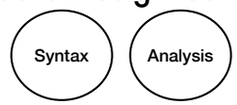
Harley Eades III

To define a programming language we first define its syntax using context-free grammars which we discussed in the previous lecture. Now we learn a new tool, inductive definitions, which are used to specify a programming languages evaluation and analysis phases including the static semantics like type checking and its dynamic semantics like abstract machines for evaluation of programs. First, we consider a familiar example...

Notes:
- Based on Pfenning's notes which I downloaded.

Balanced Parens

Core Design Concepts:



$$S \rightarrow \epsilon \mid (S) \mid SS$$

Lets consider the grammar of balanced parens that we saw in the previous lecture. Here we have a single non-terminal, S, that produces either a left-right closed set of parens, or the empty string, or concatenates two well-balanced strings of parens. We also used derivations to show when a string can be generated by this grammar...

Balanced Parens

Core Design Concepts:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow (S)S \\ &\Rightarrow ()S \\ &\Rightarrow ()(S) \\ &\Rightarrow ()() \end{aligned}$$

like we see here. But, there is a different way to present these two structures that fits how we specify and reason about structures in programming languages called inductive definitions. Let's take a look at that...

Balanced Parens

Core Design Concepts:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

Judgment

$s\ S$ read "the string s an be generated by the non-terminal S"

The grammar can be rewritten as a judgment using inference rules. A judgment is a proposition which states some property on data. Here "s S" should be read "the string s is a string that can be generated by the non-terminal S". A judgment is defined using inference rules....

Balanced Parens

Core Design Concepts:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

Judgment

$$s\ S$$

Inference Rules

$$\frac{}{\epsilon\ S}\ S_1 \qquad \frac{s\ S}{(s)\ S}\ S_2 \qquad \frac{s_1\ S \quad s_2\ S}{s_1 s_2\ S}\ S_3$$

We have three inference rules S1, S2, and S3 which correspond to each of the rules of the grammar. Let's zoom in on a rule, and try to understand it a bit better...

Balanced Parens

Core Design Concepts:

Syntax

Analysis

$$S \rightarrow \epsilon \mid ($$

$$\frac{s\,S}{(s)\,S}\ S_2$$ (if, name, then)

An inference rule is read from top-to-bottom as an if-then statement where the judgments above the line are called premises and the judgment above the line is called the conclusion. This rule states that if the string little-s can be generated by S, then adding parens around little-s can also be generated by S. Each inference rule is also given a unique name. Let's look a second rule...

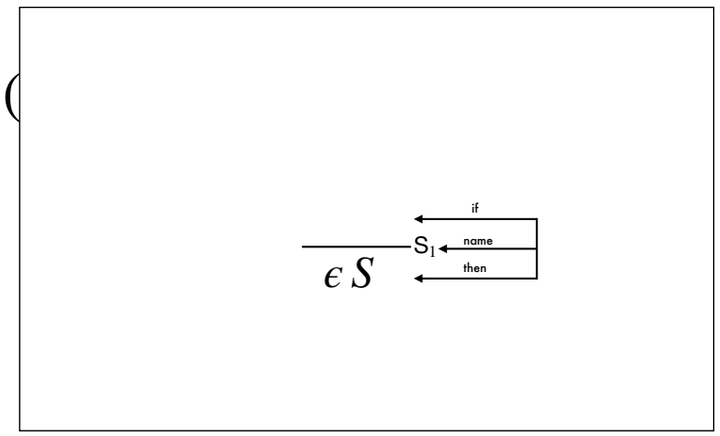
Balanced Parens

Core Design Concepts:

Syntax

Analysis

$S \rightarrow \epsilon \mid ($



This inference rule is called an axiom which has no premises above the rule. Axioms do not have any premises, and thus, are always true. This rule states that the empty string can be generated by S, and there are no premises necessary for that claim....

Balanced Parens

Core Design Concepts:

$S \rightarrow \epsilon \mid (S) \mid SS$

Judgment

$s\ S$

Inference Rules

$$\frac{}{\epsilon\ S}\ S_1$$

$$\frac{s\ S}{(s)\ S}\ S_2 \qquad \frac{s_1\ S \quad s_2\ S}{s_1 s_2\ S}\ S_3$$

The third rule has two premises, think of each premise as being separated by a conjunction, that is, if every premise is true, then the conclusion is true. So how do derivations change you might be wondering? Well...

Balanced Parens

Core Design Concepts:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

$()() \; S$

Inference Rules

$$\frac{}{\epsilon \; S}\,S_1 \qquad \frac{s \; S}{(s) \; S}\,S_2 \qquad \frac{s_1 \; S \quad s_2 \; S}{s_1 s_2 \; S}\,S_3$$

Derivations in the context-free grammar case are represented as derivations in the inference rules case. We stack rules up as they apply. This is known as the bottom-up derivation or goal directed proofs. So let's step through this. We know we need to show that this judgment is derivable. So the first thing we do is pattern-match it to the conclusion of the rules. Then apply the one that matches. Here we can see that S3 matches our conclusion, but neither S1 nor S2 do. So we apply it...

Balanced Parens

Core Design Concepts:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

Inference Rules

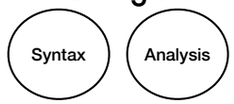
$$\frac{}{\epsilon\ S}\ S_1 \qquad \frac{s\ S}{(s)\ S}\ S_2 \qquad \frac{s_1\ S \quad s_2\ S}{s_1 s_2\ S}\ S_3$$

$$\frac{()\ S \quad ()\ S}{()()\ S}\ S_3$$

This rule splits our goal into two new subgoals. We must derive both of these new goals. So we pick one and pattern match it against the conclusions of the rules, but we can see that it only matches S2 if we make little-s the empty string, and in fact, both premises match, so we can apply it....

Balanced Parens

Core Design Concepts:



$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

$$\dfrac{\dfrac{\dfrac{}{\epsilon\, S}\,S_1}{()\, S}\,S_2 \quad \dfrac{\dfrac{}{\epsilon\, S}\,S_1}{()\, S}\,S_2}{()()\, S}\,S_3$$

Inference Rules

$$\frac{}{\epsilon\, S}\,S_1$$

$$\frac{s\, S}{(s)\, S}\,S_2 \qquad \frac{s_1\, S \quad s_2\, S}{s_1 s_2\, S}\,S_3$$

This then finishes the derivation using the S1 rule for the epsilon. A derivation is valid if and only if the derivation ends with axioms in all subgoals. Just as we see here.

Judgments and Inference Rules

Core Design Concepts:

$$\frac{J_1 \quad J_2 \quad \cdots \; J_i}{J}\ \text{name}$$

if all true

then true

Specifications

Judgements, here J, and inference rules are really useful and expressive at specifying structures in programming languages, but what about algorithms? For example, our context-free grammar examples gives us a way of specifying grammars and their derivations, but what about a parsing algorithm? We can in fact build algorithms that derive judgments using our inference rules. This inductive definition to implementation is known as Logic Programming! Let's consider an example of a parsing algorithm for our balanced parens example.

Logic Programming: Parsing

Core Design Concepts:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

Judgment

$$s\ S$$

Inference Rules

$$\frac{}{\epsilon\ S}\ S_1 \qquad \frac{s\ S}{(s)\ S}\ S_2 \qquad \frac{s_1\ S \quad s_2\ S}{s_1 s_2\ S}\ S_3$$

$$S \Rightarrow SS \Rightarrow \epsilon S \Rightarrow \epsilon\epsilon = \epsilon$$
$$S \Rightarrow \epsilon$$

First, this grammar is ambiguous! Here are two left-most derivation of epsilon. There are other ones as well...

Logic Programming: Parsing

Core Design Concepts:

Judgment

$s\ S$

Inference Rules

$$S \to \epsilon \mid (S) \mid SS$$

$$\frac{}{\epsilon\ S}\ S_1 \qquad \frac{s\ S}{(s)\ S}\ S_2 \qquad \frac{s_1\ S \quad s_2\ S}{s_1 s_2\ S}\ S_3$$

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()$$
$$S \Rightarrow (S) \Rightarrow ()$$

Here are two left-most derivations of open-close parens. As we discussed in the previous lecture in order for a parsing algorithm to work effectively we need an to ensure there are at-most one derivation for every string in the language of the grammar. The good news is that usually we can modify our grammar to produce one that is unambiguous...

Logic Programming: Parsing

Core Design Concepts:

Judgment

$s\ L$

Inference Rules

$$\frac{}{\epsilon\ L}\ \mathrm{L}_1 \qquad \frac{s_1\ L \quad s_2\ L}{(s_1)s_2\ L}\ \mathrm{L}_2$$

$$L \rightarrow \epsilon \mid (L)L$$

$$L \Rightarrow (L)L \Rightarrow ()L \Rightarrow ()$$
$$L \Rightarrow \epsilon$$

This grammar removes the ambiguity and we can see here that these derivations are the only ones that exist for the strings open-close parens and empty string. In fact, we can prove...

Logic Programming: Parsing

Core Design Concepts:

$$L \rightarrow \epsilon \mid (L)L$$

Judgment

$$s\,L$$

Inference Rules

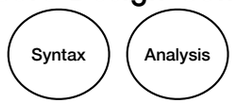
$$\frac{}{\epsilon\,L}\,L_1 \qquad \frac{s_1\,L \quad s_2\,L}{(s_1)s_2\,L}\,L_2$$

$s\,S$ if and only if $s\,L$

That if a string is derivable using our previous judgment then it is still derivable using the new judgment and vice versa. We would prove this using a technique called rule induction, but we will not go into the details of this now. Now let's build a parsing algorithm for our new judgment.

Logic Programming: Parsing

Core Design Concepts:



Judgment

$$s\ L$$

Inference Rules

$$\frac{}{\epsilon\ L}\ L_1 \qquad \frac{s_1\ L \quad s_2\ L}{(s_1)s_2\ L}\ L_2$$

$k \vdash s$ "s is a string of balanced parens with respect to counter k"

$$\frac{}{0 \vdash \epsilon}\ P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\ P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\ P_D$$

Here we define a new judgment and define its inference rules. The judgment states that the string s has a balanced set of parens with respect to the counter k. Here we are going to keep a counter that when we see an open paren we will increment the counter, but when we see a closing paren we will decrement the counter. If the string is balanced then we will add 1 for each open paren and subtract 1 for every closing paren, and thus, when we reach the empty string we will be at 0. This idea is encapsulated by these rules. The first states that the empty string is valid only when the counter is 0, the second rule P-sub-I (I here is for increment) says that if the string open-paren s is valid with counter k, then s must be valid in counter k+1. Here I'm reading these as if we are applying the rules, that is, in a goal-directed way. The decrement rule, P-sub-D, says that if close-paren s is valid with counter k, then s must be valid with counter k-1, but we must ensure that k is bigger than 0 here or we would accept strings that are unbalanced. Let's take a quick look at an example....

Core Design Concepts:

Syntax Analysis

Logic Programming: Parsing

Judgment

$$k \vdash s$$

Inference Rules

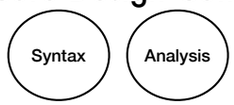
$$\frac{}{0 \vdash \epsilon}\, P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\, P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\, P_D$$

$$0 \vdash (())$$

So we want to parse the string shown here: open-open-close-close. We start with a counter of 0, because we haven't inspected any tokens yet. Next I pattern match this goal to the conclusions of the rules, and we can see that only one rule applies: the increment rule...

Logic Programming: Parsing

Core Design Concepts:



Judgment

$$k \vdash s$$

Inference Rules

$$\frac{}{0 \vdash \epsilon}\,P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\,P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\,P_D$$

$$\frac{\dfrac{1 \vdash ())}{0 \vdash (())}\,P_I}{}$$

now our new goal is to show that open-open-close is valid with counter 1. So we pattern match again, and see that the increment rule is still the only one that applies...

Logic Programming: Parsing

Core Design Concepts:

Judgment

$k \vdash s$

Inference Rules

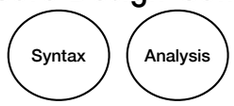
$$\frac{}{0 \vdash \epsilon}\,P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\,P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\,P_D$$

$$\dfrac{\dfrac{\dfrac{}{2 \vdash))}\,P_I}{1 \vdash ())}\,P_I}{0 \vdash (())}$$

At this point we have consumed all of the open-parens and are now required to consume the closing parens. Pattern matching our goal to the conclusions of the rules we can see that the decrement rule is the only one that applies, but we do have to make sure that our counter is non-zero, and we can see that it is....

Logic Programming: Parsing

Core Design Concepts:



Judgment

$k \vdash s$

Inference Rules

$$\frac{}{0 \vdash \epsilon}\,P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\,P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\,P_D$$

$$\dfrac{\dfrac{\dfrac{1 \vdash) \qquad 2 > 0}{2 \vdash))}\,P_D}{1 \vdash ())}\,P_I}{0 \vdash (())}\,P_I$$

And, pattern matching again, we can see that we must apply the decrement rule, and the counter is bigger than 0...

Logic Programming: Parsing

Core Design Concepts:

Judgment

$k \vdash s$

Inference Rules

$$\frac{}{0 \vdash \epsilon}\,P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\,P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\,P_D$$

$$\dfrac{\dfrac{\dfrac{\dfrac{0 \vdash \epsilon \quad 1 > 0}{1 \vdash) \quad 2 > 0}\,P_D}{2 \vdash))}\,P_I}{1 \vdash ())}\,P_I}{0 \vdash (())}$$

Finally, we can see that we can conclude this derivation using the P-epsilon rule because we have reached the empty string with a counter of 0....

Logic Programming: Parsing

Core Design Concepts:

Judgment

$k \vdash s$

Inference Rules

$$\frac{}{0 \vdash \epsilon}\,P_\epsilon \qquad \frac{k+1 \vdash s}{k \vdash (s}\,P_I \qquad \frac{k-1 \vdash s \quad k > 0}{k \vdash)s}\,P_D$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \vdash \epsilon}\,P_\epsilon \quad 1 > 0}{1 \vdash)}\,P_D \quad 2 > 0}{2 \vdash))}\,P_D}{1 \vdash ())}\,P_I}{0 \vdash (())}\,P_I$$

Notice that we followed the same operation constructing the derivation for our parsing judgment as we did for the context-free grammar derivation. We have a judgment whose inference rules have unique conclusion, and so to obtain an algorithm for finding derivations, we simply pattern match on the conclusion, apply the rule, and check any side conditions on the rule. If we reach an axiom then we were successful. Notice here that we only get one derivation for every conclusion! Logic programming languages have its users specify the inference rules, and automatically generates programs based on derivations of the rules using pattern matching like we see here. We will do some more examples in class.