Lecture: Syntax, Inference Rules, and Our First Example PL
Programming Languages (CSCI 3300)

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

1 Introduction to the course

Programming languages (PLs) are ubiquitous in computer science. Any student of computation has used some programming language whether they a computer scientist, software engineer, or information technologist. This then implies that the PLs are the most important artifacts in the computational sciences.

One would find it hard to argue that PLs are important, but one could argue that we already have a lot of PLs, and it seems as if these are enough, but is this true? I claim that it is not the case. Computational devices are ubiquitous in our society. They power our power plants, our water supply, virtually every consumer product is built using some computational device, in addition they power our automobiles, our air planes, and our medical devies. It is safe to say that we put our vary lives in the hands of computational devices, which implies that we put our lives in the hands of the programmers who designed the software systems powering such devies. What happens when a software bug finds its way into the safety critical devices that we depend on so dearly?

In 2010 there were a approximately a hundred reports made to the National Highway Traffic Safety Administration of potential problems with the braking system of the 2010 Toyota Prius [1]. The problem was that the anti-lock braking system would experience a "short delay" when the brakes where pressed by the driver of the vehicle [4]. This actually caused some crashes. **Toyota found that this short delay was the result of a software bug, and was able to repair the the vehicles using a software update** [3]. Another incident where substantial harm was caused was in 2002 where two planes collided over Überlingen in Germany. A cargo plane operated by DHL collided with a passenger flight holding fifty-one passengers. Air-traffic control did not notice the intersecting traffic until less than a minute before the collision occurred. Furthermore, the on-board collision detection system did not alert the pilots until seconds before the collision. It was officially ruled by the German Federal Bureau of Aircraft Accidents Investigation that the **on-board collision detection software was indeed faulty** [2].

Avi Rubin is a professor at John Hopkins university and has recently shown that pacemakers can be remotely hacked. It may come to the surprise to computer scientists, but pacemakers have wifi access now. Avi has shown that this connection can be indeed hacked, and could be used by the attacker to cause a fatal heart attack. These hacks are indeed due to software vulnerabilities – that is bugs.

How can we prevent these disastrous problems? Currently, we using testing. Programmers come up with a huge number of test inputs to a program, and then they run the program on each one of these tests. If the program behaves across all the tests, then we consider the program as being correct. Does this catch all bugs? No! In fact, no amount of testing could ever **prove** the absence of all bugs! If the programmer is not clever enough, then they could indeed miss some corner case. This is evident in the examples above. So is it possible to be sure a program contains no bugs? The only way to be sure a program contains no bugs is by mathematically proving that it meets some specification. Every programmer uses some criteria to write a program, and a programs specification corresponds to this criteria transformed into a list of mathematical formulas. Then if one can prove that these formulas are all true with respect to the program, then one can be sure with mathematical certainty that it is correct. That is bug free. The joy of this approach is that if one of the formulas cannot be proven, then a bug exists. Furthermore, this implies that the bug would be found during development instead of in production. So for example, the Toyota Prius bug in the

breaking system would have been found before all those crashes, and costly recalls. So why is it the case that companies use testing instead of mathematical proof? The reason is that current programming languages are not amendable to mathematical proofs. There are just to many complex interactions.

Lets consider an example. Suppose I say a function `squared` takes in an `int` and outputs an `int`, then we know something about `squared`, we know that if we take and apply it to, say, `2`, then we know that it will give us back an `int`, perhaps, `4`. Is it guaranteed that this is all it does? Suppose we are given the `squared` function and it is written in C# do we know that it does nothing more than compute some integer? The answer is most definitely no. In C# the function `squared` could ask for some user input, print data to the screen, and even make some network transactions or ask for the time of day. Hidden effects like these are all called **side effects**. So in a programming language like C#, C, C++, Java, PHP, Python, and many more mainstream PLs we cannot really be sure what `squared` is doing. Thus making it very difficult to reason about these programs. So can we fix this problem? Can we design a PL that is more amendable to mathematical reasoning?

I claim – as do many other PL researchers – that we can prevent software bugs by mathematical reasoning by adopting new more rigorously studied programming languages. In fact, designing these types of PLs is the very driving force of modern day PL research. To design these types of programming languages we must revisit the very foundations of programming languages, and rebuild from the ground up. It turns out that by adopting the notion of a pure strongly typed PL we can rule out a large number of software bugs during development without using testing, and before lives are taken!

A pure strongly typed PL is one in which there are either no side effects at all or all side effects are accounted for in the type system. Consider the `squared` function again. The type of this function can be written as `int` $\rightarrow$ `int` which means taken an `int` as input, and then return an `int` as output. Now suppose `squared` prints out some data to the screen, then in a pure strongly typed PL the type becomes `int` $\rightarrow$ `IO(int)`. Here `IO(int)` means the function returns an `int` and conducts some input-output effect. This allows us to take into account side effects when reasoning about the program. So how does all this work? Well that is part of the topic of this course.

Another part of the major topic of this course is on pure strongly typed **functional** PLs. Functional programming languages differ from the PLs like C# or Java by having a mathematical foundation. The sole datatype is a function, and in fact, the only thing a functional PL contains are functions. It turns out that we can do a lot with this paradigm. PLs like C#, Java, and C are known as imperative and object oriented languages. These have a foundation in an actual machine like a Turing Machine, while, functional PLs have their foundation in mathematics. However, it turns out that they are equivalent in power. One major part of this course is to understand the design and implementation of functional PLs. Functional programming languages are also great for exploring different design features, and we will use them to study features like parametric polymorphism, and recursion. By the end of this course each student will understand how different each PL can be, and be able to better differentiate between the various PLs when choosing one for a particular project.

So a natural question is then is this all theoretical? What baring does any of this have on the real world? A lot! The industry is already taking steps to push the old style of programming out, and adopt new PLs that are highly motivated by the pure strongly typed functional PLs. A great example is Apple's new PL called Swift. They have decided to completely abandon all Objective C code from all their IOS devices. Swift is based on strongly typed functional programming because of its brevity of code, and the strong correctness guarantees. Another example, is Mozilla's Rust programming language which is a systems programming language which is designed for speed. A final example is the Haskell programming language. Haskell has paved the way for functional programming language research and much of the features of Swift and Rust where first studied in Haskell, and Haskell is used in the industry, especially, the financial sector. Lastly, Haskell is the last major topic of this course.

2 Grammars

The human interface to computational devices are PLs, but the interface to PLs is syntax. So how to we formally define the syntax of programming languages? We use a mathematical device called a grammar. Now instead of going into the fascinating area of formal language theory – which is where grammars were studied extensively – we will instead focus on several examples to learn how grammars work.

We can consider the syntax of programming languages to represent a formal language. A formal language is a set words generated from a set of symbols. Now consider the question, how can we generate all of these words? This is the very goal of a grammar.

Consider the set $L_{\text{even}} = \{aa, aaaa, aaaaaa, aaaaaaaa, aaaaaaaaaa, aaaaaaaaaaaa, \ldots\}$. This set can be more compactly described as $\{w \mid w = a^{2(n+1)} \text{ for } n \in \mathbb{N}\}$ where $\mathbb{N}$ is the set of natural numbers, or the set $\{0, 1, 2, 3, 4, 5, 6, \ldots\}$. So as we discussed above, the set L_{even} is a language, but what language? It is the language where all words are generated from the single symbol a, and where each word contains an even number of a's. Now a grammar for this language looks like the following:

S	$\rightarrow$	aE
E	$\rightarrow$	a
E	$\rightarrow$	aO
O	$\rightarrow$	aE

A grammar consists of one or more productions denoted $A \rightarrow s$ where we call A the non-terminal of the production. Note that the lefthand side of a production is always a non-terminal, and not a composition of terminals and non-terminals. Each production generates either a symbol in the language – also called a non-terminal – or is the combination of terminals (symbols of the language) and non-terminals. So the righthand side of a production is a composition of non-terminals and terminals, but it may also only consist of non-terminals. In the example above I have put boxes around each production, colored the non-terminals red, and the terminals blue. The non-terminal S is a special non-terminal called the start non-terminal. The start non-terminal is always used first no matter what. A grammar may also have several productions associated with the start non-terminal.

Once we have a grammar we can use it to generate any word in its corresponding language. We generate words using a grammar by constructing a derivation. Lets consider an example:

$$\begin{aligned}
S &\Rightarrow aE \\
&\Rightarrow aaO \\
&\Rightarrow aaaE \\
&\Rightarrow aaaaO \\
&\Rightarrow aaaaaE \\
&\Rightarrow aaaaaaO \\
&\Rightarrow aaaaaaaE \\
&\Rightarrow aaaaaaaaO \\
&\Rightarrow aaaaaaaaaE \\
&\Rightarrow aaaaaaaaaa
\end{aligned}$$

A derivation always begins with one of the productions whose lefthand side is the start symbol. Then we proceed by simply choosing one of the non-terminals on the lefthand side – it does not matter which is chosen or in what order – and replacing it with the lefthand side of one of its associated productions. We repeat this until all non-terminals have been replaced by terminals. So every derivation ends with a word in the language the grammar generates.

This is great, but now ask the question, what if we are given a word, say, this word

$$aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$$

, is it possible to test to see if it is in the language L_{even}? Now for our simple example we could just tally up the number of a's and if its even, then accept, otherwise we reject the above word, but if this were a more complex language, like say, a PL, then that would not work. It turns out that we can say, well, if there exists a derivation that ends with the above word, then it is a member of the language L_{even} otherwise it is not. In fact, there are known algorithms for doing just that. In fact, this is the basic operation for parsers, and when using a PL you obtain a syntax error, the compiler took the input and tried to derive using the grammar of the PL. So grammars are the driving force behind syntax, and parser technology.

Lets consider another more interesting example:

$$\begin{array}{lcl} S & \to & () \\ S & \to & (S) \\ S & \to & SS \end{array}$$

What language does this grammar generate? Its the language of balanced parenthesis. A few example words would then be (), ()(), (((((((()))))))), and (((((((()))))))()(). However, these are not words in the language generated by the above grammar: (,)(, and ((((())))).

I would like to mention one notational convention I am using. Notice that if I want to have multiple productions for a symbol, like S in the previous example, I write multiple productions with the same non-terminal on the lefthand side. There is a shorthand convention that we will use where we combine these all into a single production. For example, the previous grammar can be written like this:

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

Here the | symbol should be read as "or."

Finally, lets see how we can put all this together to formally specify the syntax of a programming language. The following grammar is a grammar of a toy programming language:

$$\begin{array}{rl} \text{(booleans)} & b \to \mathsf{true} \mid \mathsf{false} \mid b \wedge b \mid b \vee b \mid \neg b \mid e > e \\ \text{(expressions)} & e \to 0 \mid e+1 \mid e * e \mid e + e \mid e - e \\ \text{(command)} & S \to \mathsf{skip} \mid x := e \mid \mathsf{if}\, b\, \mathsf{then}\, S\, \mathsf{else}\, S \mid \mathsf{while}\, b\, \mathsf{do}\, S \end{array}$$

This is slightly more complex, but the same rules we have been discussing above hold. An example derivation is:

$$\begin{array}{rcl} S & \Rightarrow & \mathsf{if}\, b\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, e > e\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, e+1 > e\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > e\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > e+1\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > e+1+1\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > 0+1+1\, \mathsf{then}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > 0+1+1\, \mathsf{then}\, \mathsf{while}\, b\, \mathsf{do}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > 0+1+1\, \mathsf{then}\, \mathsf{while}\, \mathsf{true}\, \mathsf{do}\, S\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > 0+1+1\, \mathsf{then}\, \mathsf{while}\, \mathsf{true}\, \mathsf{do}\, \mathsf{skip}\, \mathsf{else}\, S \\ & \Rightarrow & \mathsf{if}\, 0+1 > 0+1+1\, \mathsf{then}\, \mathsf{while}\, \mathsf{true}\, \mathsf{do}\, \mathsf{skip}\, \mathsf{else}\, \mathsf{skip} \end{array}$$

At this point we have seen basically all that we will need to know about grammars for this course and the large majority of PL research. However, the previous grammar we saw for the toy PL is a bit more formal than we will need.

So we now look at an alternative presentation of the same grammar:

$$\begin{array}{rl} \text{(booleans)} & b ::= \mathsf{true} \mid \mathsf{false} \mid b \wedge b \mid b \vee b \mid \neg b \mid e > e \\ \text{(expressions)} & e ::= 0 \mid e+1 \mid e * e \mid e + e \mid e - e \\ \text{(command)} & c ::= \mathsf{skip} \mid x := e \mid \mathsf{if}\, b\, \mathsf{then}\, c\, \mathsf{else}\, c \mid \mathsf{while}\, b\, \mathsf{do}\, c \end{array}$$

This is called the Backus-Naur Form (BNF) of the formal grammar above. In PL we usually do not indicate a start non-terminal, but rather we give the grammar as a means of understanding how to derive legal expressions of the language.

3 Iffy-Lang

In this section I would like to introduce the PL Iffy-Lang. Its sytax is as follows:

$$\text{(Bool)} \quad b \quad ::= \quad 1 \mid 0 \mid b \;\&\; b \mid b \mid b \mid \mathsf{if}\, b \,\mathsf{then}\, b \,\mathsf{else}\, b$$

Now the above syntax allows us to write small programs like:

$$\mathsf{if}\, b \,\mathsf{then}\, 0 \,\mathsf{else}\, 1$$

However, the syntax does not describe how to actually compute with such programs. So how do we run them?

The operational behavior of a program is often described by a set of rules called the reduction rules. The following rules describe the operational behavior of Iffy-Lang:

$$\frac{}{(1 \;\&\; 1) \rightsquigarrow 1}\ \textsc{andTrue} \qquad \frac{}{(0 \;\&\; 1) \rightsquigarrow 0}\ \textsc{andFalse1} \qquad \frac{}{(1 \;\&\; 0) \rightsquigarrow 0}\ \textsc{andFalse2}$$

$$\frac{}{(0 \;\&\; 0) \rightsquigarrow 0}\ \textsc{andFalse} \qquad \frac{b_1 \rightsquigarrow b_1'}{(b_1 \;\&\; b_2) \rightsquigarrow (b_1' \;\&\; b_2)}\ \textsc{and1} \qquad \frac{b_2 \rightsquigarrow b_2'}{(b_1 \;\&\; b_2) \rightsquigarrow (b_1 \;\&\; b_2')}\ \textsc{and2}$$

$$\frac{}{(1 \mid 1) \rightsquigarrow 1}\ \textsc{orTrue} \qquad \frac{}{(0 \mid 1) \rightsquigarrow 1}\ \textsc{orFalse1} \qquad \frac{}{(1 \mid 0) \rightsquigarrow 1}\ \textsc{orFalse2}$$

$$\frac{b_1 \rightsquigarrow b_1'}{(b_1 \mid b_2) \rightsquigarrow (b_1' \mid b_2)}\ \textsc{or1} \qquad \frac{b_2 \rightsquigarrow b_2'}{(b_1 \mid b_2) \rightsquigarrow (b_1 \mid b_2')}\ \textsc{or2} \qquad \frac{}{(0 \mid 0) \rightsquigarrow 0}\ \textsc{orFalse}$$

$$\frac{}{(\mathsf{if}\, 1 \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2) \rightsquigarrow b_1}\ \textsc{ifTrue} \qquad \frac{}{(\mathsf{if}\, 0 \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2) \rightsquigarrow b_2}\ \textsc{ifFalse}$$

$$\frac{b \rightsquigarrow b'}{(\mathsf{if}\, b \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2) \rightsquigarrow (\mathsf{if}\, b' \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2)}\ \textsc{if1} \qquad \frac{b_1 \rightsquigarrow b_1'}{(\mathsf{if}\, b \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2) \rightsquigarrow (\mathsf{if}\, b \,\mathsf{then}\, b_1' \,\mathsf{else}\, b_2)}\ \textsc{if2}$$

$$\frac{b_2 \rightsquigarrow b_2'}{(\mathsf{if}\, b \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2) \rightsquigarrow (\mathsf{if}\, b \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2')}\ \textsc{if3}$$

One should read these rules from the top down as an "if-statement", that is, the rule:

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

should be read as if $P_1, \cdots, P_n$ hold, then C holds. So for example the rule:

$$\frac{b \rightsquigarrow b'}{(\mathsf{if}\, b \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2) \rightsquigarrow (\mathsf{if}\, b' \,\mathsf{then}\, b_1 \,\mathsf{else}\, b_2)}\ \textsc{if}$$

says, if $b \rightsquigarrow b'$ holds, then $(\mathsf{if}\, b\, \mathsf{then}\, b_1\, \mathsf{else}\, b_2) \rightsquigarrow (\mathsf{if}\, b'\, \mathsf{then}\, b_1\, \mathsf{else}\, b_2)$ holds.

Now the symbol $\rightsquigarrow$ should be read as "reduces to", hence, we read $b \rightsquigarrow b'$ as "b reduces to b'". The rules above can be seen as defining what it means for a boolean b to reduce to b'.

The previous set of inference rules tell us how to reduce an Iffy program for one step only. The following rules generalize this to allow for multiple steps:

$$\frac{}{b \rightsquigarrow^* b}\ \textsc{Zero} \qquad \frac{b \rightsquigarrow b'}{b \rightsquigarrow^* b'}\ \textsc{Step} \qquad \frac{b1 \rightsquigarrow^* b2 \qquad b2 \rightsquigarrow^* b3}{b1 \rightsquigarrow^* b3}\ \textsc{Mult}$$

The judgment $b \rightsquigarrow^* b'$ is called multi-step reduction.

References

[1] blogs.consumerreports.org. Consumer reports cars blog: Japan investigates reports of prius brack problem, 2010.

[2] The German Federal Bureau of Aircraft Accidents. Investigation report, 2004.

[3] Reuters. Toyota to recall 436,00 hybrids globally-document, February 2010.

[4] thedetroitbureau.com. Nhtsa memo on regenerative braking, April 2011.