Comp 411
Principles of Programming Languages
Lecture 2
Syntax

Corky Cartwright

January 11, 2023

Syntax: The Boring Part of Programming Languages

- Programs are represented by sequences of symbols (not characters).
- These symbols are represented as sequences of characters that can be typed on a common keyboard (ASCII).
- What about Unicode? (Ignored in this course but potentially important in practice.)
- To analyze or execute the programs written in a language, we must translate the ASCII/Unicode representation for a program to a higher-level tree representation. This process, called *parsing*, conveniently breaks into two parts:
  - *lexical analysis* (sometimes called *lexing* or *tokenization*), and
  - *context-free parsing* (often simply called *parsing*).

Lexical Analysis

- Consider this sequence of characters: `begin middle end`
- What are the smallest meaningful pieces of syntax in this phrase?
- The process of converting a character stream into a corresponding sequence of meaningful symbols (called *tokens* or *lexemes*) is called *tokenizing*, *lexing* or *lexical analysis*. A program that performs this process is called a *tokenizer* or a *lexer*.
- In Scheme/Racket, we tokenize
  `(set! x (+ x 1))` as
  `(` `set!` `x` `(` `+` `x` `1` `)` `)`
- Similarly, in Java, we tokenize
  `System.out.println("Hello World!");` as
  `System` `.` `out` `.` `println` `(` `"Hello World!"` `)` `;`

Lexical Analysis, cont.

- Tokenizing is straightforward for most languages because it can be performed by a finite automaton (equivalent to a regular grammar for those of you who have taken 412 or 481) that matches the longest possible string of characters as the next token. Fortran is an interesting exception!
- The rules governing this process are (a very boring) part of the language definition. They typically can be formulized as a regular grammar.
- The details are generally provided as part of a language definition but subsequently glossed over as uninteresting.
- Parsing a stream of *tokens* (symbols) into structural description of a program (typically a tree) is harder.

Parsing

- Consider the Java statement: `x = x + 1;`
  where `x` is an `int` variable.
- The grammar for Java stipulates (among other things):
  - The assignment operator `=` may be preceded by an identifier (other more complex, possibilities exist as well) and must be followed by an expression.
  - An expression may be two expressions (restricted for ease of parsing to special kinds of expressions) separated by a binary operator such as `+`.
  - An assignment expression can serve as a statement if it is followed by the statement terminator symbol `;`. Hence, we can deduce from the grammatical rules of Java that the above sequence of characters (tokens) is a legal program statement (provided `x` has been declared) that performs an assignment.
- The rules for constructing/recognizing sequences of tokens that are "well-formed" programs are typically specified using a Context Free Grammar (CFG) If you are unfamiliar with Context Free Grammars, look up the topic on Wikipedia.

Parsing Token Streams into Trees

- Consider the following ways to express an assignment operation:

```
x = x + 1            [Java]
x := x + 1           [Algol]
(set! x (+ x 1))     [Scheme]
```

- Which of these do you prefer? It should not matter much.
- To eliminate the irrelevant syntactic details, we can create a streamlined data representation that represents program syntax using trees. Each language construct and program operation is represented by a tree node. The leaves of the tree are typically language constants. For instance, the abstract syntax for the assignment code given above could be (assuming Scheme as the *implementation* language)

```
(make-assignment <Rep of x> <Rep of x + 1>)
```

- Or (in Java as the implementation language)

```
new Assignment(<Rep of x> , <Rep of x + 1>)
```

A Simple Example

Assume we are given the (extended) *context-free grammar* CFG with only one production:

```
Exp ::= Num | Var | (Exp Exp) | (lambda Var Exp)
```

where

`Num` is the set of numeric constants (given in the lexer specification)

`Var` is the set of variable names (given in the lexer specification)

To represent this syntax as trees (abstract syntax) in Scheme/Racket

```
; exp := (make-num number) | (make-var symbol) | (make-app exp exp) |
;        (make-proc symbol exp)
(define-struct (num n)) ;; num is the constructor name, n is a field
(define-struct (var s))
(define-struct (app rator rand))
(define-struct (proc param body))  ;; param is a field name not a var
```

where an **`app`** structure represents a function application and a **`proc`** structure represents a function definition (typically a lambda-abstraction). Structures in Scheme correspond to structures in C/C++ and data classes in Java.

Top Down (Predictive) Parsing

- Idea: design the grammar so that we can always tell what rules can be used next starting from the root of the parse tree by looking ahead (in a left-to-right scan) some small number (*k*) of *tokens* (formally LL(*k*) parsing in the context of a *context-free grammar* defining the set of legal programs)
- This algorithm an easily be implemented by manual coding using a technique called *recursive descent.*
- Conceptual aid: we use *syntax diagrams* to express the legal sequences of symbols that appear in production rules. Syntax diagrams are (almost) formally equivalent to context free grammars but implicitly describe a simple recursive parsing strategy (recursive descent) if path branching can be resolved by look-ahead. They are some small but important technical differences between syntax diagrams and extended context-free grammas which are generally ignored in the literature.

Top Down (Predictive) Parsing cont.

- The intuition behind syntax diagrams is program *recognition* (parsing) while the intuition behind context-free grammars is program *generation*. A key example where these two formalizations disagree is `if` statements with optional `else` clauses. The extended CFG formulation is ambiguous (which `if` does a specific `else` match?) while the syntax diagram formulation is not (because of the *maximal matching restriction* in the recognition process).
- Intuition: *k*-symbol look-ahead is used to determine which branch to take at a fork in a syntax diagram.
- We try to design LL(*k*) grammars (and the corresponding syntax diagrams) so that *k* is ≤ 1. The precise definition of LL(*k*) is subtle; if a parser can decide which branch (to take at a branching point in a syntax diagram using the next symbol in the input is LL(0) not LL(1). Looking at the next symbol to determine which branch to take is not classified as looking ahead.
- Reference: see http://www.bottlecaps.de/rr/ui

Example: Jam Syntax

- Jam is the toy functional language that we will use throughout the course. You may be surprised by the richness of the mathematical structure underlying such a simple language.
- Look at the PDF File containing syntax diagrams for Jam:
  b
  https://www.cs.rice.edu/~javaplt/411/23-spring/Assignments/1/RevisedSyntaxDiagrams.pdf

  or https://github.com/RiceComp411/Assignment-1-Master-2023/blob/main/SyntaxDiagrams.pdf
- Reference: see **http://www.bottlecaps.de/rr/ui**