COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 5: A Simple Compiler

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

February 3, 2022

Review

Last time, we got our feet wet generating parser with BNFC, and saw how we can hook into the BNFC generated code and use it's parsing facilities. Homework 1 deadline has officially passed, so hopefully you got a good chance to use some of the things we've talked about so far.

Today, we'll start talking about the things that will be helpful for completing the next homework, which will be the first compiler we'll make in this course.

A Simple Imperative Language

To guide our discussion today, let's introduce a very simple imperative language with variable assignment, arithmetic and logical expressions, if-statements, and basic looping.

This will be the first language we use that has multiple non-terminals, and we'll see how that affects the way we traverse the abstract programs.

Arithmetic Expressions

We will only allow variables to be given numeric values, and we'll include addition, subtraction, and multiplication in our language. Here's the arithmetic expression in our grammar:

```
AVar. AExp3 ::= Ident ;
AInt. AExp3 ::= Integer ;
AMul. AExp2 ::= AExp2 "*" AExp3 ;
AAdd. AExp1 ::= AExp1 "+" AExp2 ;
ASub. AExp  ::= AExp "-" AExp1 ;

coercions AExp 3 ;
```

Boolean Expressions

We have logical not, or, and operators and some basic equality checks.

```
BTru. BExp2 ::= "true" ;
BFls. BExp2 ::= "false" ;
BNot. BExp2 ::= "!" BExp2 ;
BEq.  BExp1 ::= AExp "==" AExp ;
BLeq. BExp1 ::= AExp "<=" AExp ;
BOr.  BExp  ::= BExp1 "||" BExp ;
BAnd. BExp  ::= BExp1 "&&" BExp ;

coercions BExp 2 ;
```

Statements

A program in our language will be a list of *statements*. Statements are built up from the expressions in our language:

```
SInit.  Stm ::= Ident ":=" AExp ";" ;
SAss.   Stm ::= Ident "=" AExp ";" ;
SBlock. Stm ::= "{" [Stm] "}" ;
SIf.    Stm ::= "if" BExp "then" Stm "else" Stm ;
SWhile. Stm ::= "while" BExp "do" Stm ;
SPrint. Stm ::= "print" AExp ";" ;
```

Notice that statements don't have/need precedence or associativity because they are unambiguous as it is.

Programs

Finally we have our rule for programs (the start symbol). A program is either a statement followed by another program, or empty. We can encode that in LBNF as follows:

```
Prog.  Imp ::= Stm Imp ;
ProgE. Imp ::= ;
```

However, LBNF also has a nice built-in syntax for lists that makes this a little cleaner:

```
Prog. Imp ::= [Stm] ;
separator Stm "" ;
```

Programs

```
Prog. Imp ::= [Stm] ;
separator Stm "" ;
```

This says a program is a list of statements, and lists of statements do not have a separator.

Transpile to JavaScript

Now, we'd like to be able to run these programs. That's a whole lot of work though! Lots of decisions to make and state to keep track of... instead let's just transpile it to JavaScript and let Node do all the work for us.

Hinting Towards Semantics

In writing this transpiler, we will be forced to think about what the statements and expressions in our language really mean and what effects they should have. This meaning is referred to as the semantics of the language.

If we are going to translate/compile/transpile our language into JavaScript, one of our goals should be to preserve the semantics of our source language in the generated target. So we not only need to think about the semantics of our little language, but also the semantics of JavaScript, and convince ourselves that they are equivalent.

Transpiling Statements

We can approach this from several different directions, but let's start with a simple program, and think about what the equivalent JavaScript program should be:

```
x := 5;
x = 10;
print x;
```

Transpiling Statements

We can approach this from several different directions, but let's start with a simple program, and think about what the equivalent JavaScript program should be:

```
x := 5;
x = 10;
print x;
```

```
var x = 5;
x = 10;
console.log(x);
```

Transpiling Arithmetic Expressions

What about more complex statements, with more complex arithmetic expressions?

```
i := 1;
y := 1;
while i <= 10 do {
  y = y * i;
  i = i + 1;
}
```

Transpiling Arithmetic Expressions

What about more complex statements, with more complex arithmetic expressions?

```
i := 1;
y := 1;
while i <= 10 do {
  y = y * i;
  i = i + 1;
}
```

```
var i = 1;
var y = 1;
while (i <= 10) {
  y = y * i;
  i = i + 1;
}
```

Very straightforward! In this case the semantic translation is simple because our source language semantics are very simple. Let's implement it.

Implementing the Transpiler

Live demo of the transpiler

Homework 2 Discussion

Let's talk about how this relates to homework 2...

Real World Transpilers

Tons of compilers that target JavaScript exist. Wikipedia page about source-to-source compilers also includes a list of the most popular transpilers.

The End