Languages: Compilers and interpreters

Glenn Bruns
CSUMB

Learning outcomes

After this lecture, you should be able to:

- define 'syntax tree'
- write code to create a syntax tree during parsing
- explain the difference between a compiler and an interpreter

Recap

So far we have:

- defined syntax with a BNF grammar
- transformed the grammar
- written a predictive parser from the transformed grammar

but: we want to do more than syntax checking

Building a syntax tree

Often you want to build a syntax tree while you parse.

The syntax tree can be used for lots of things:

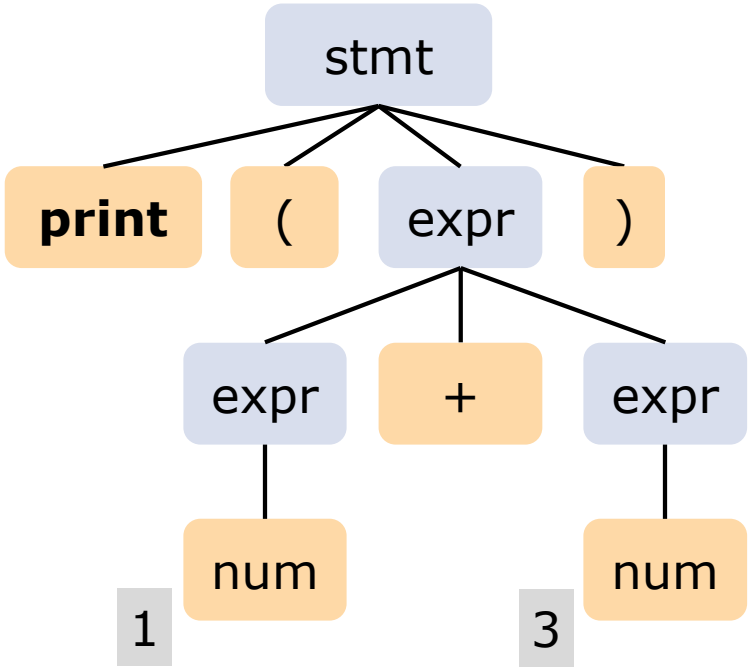
- "pretty-printing"
- compiling
- interpreting
- optimizing
- type-checking
- …

Parse tree vs. syntax tree

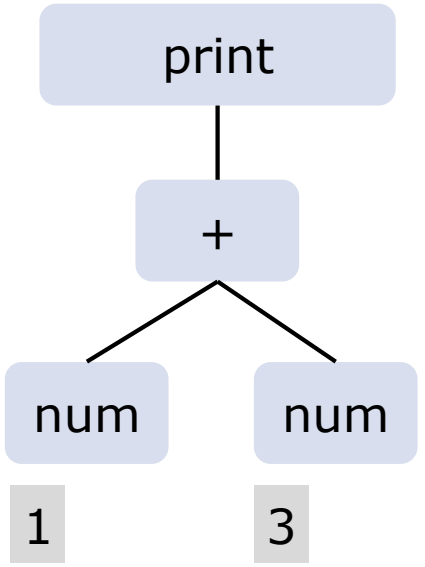
```
stmt ::= print ( expr )
       | id = expr
expr ::= id | expr + expr
```

In an abstract syntax tree, nodes are operators and children of a node are the operands.

parse tree:



abstract syntax tree:

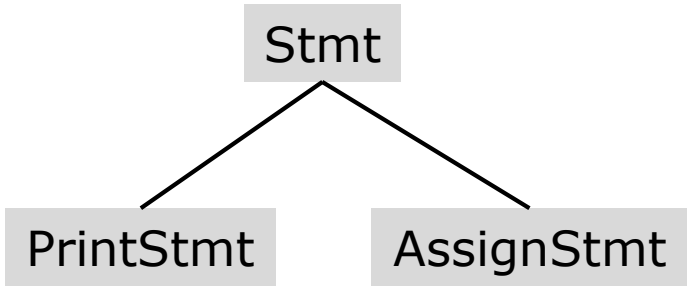


Program as Data Structure

```
stmt ::= print ( expr )
       | id = expr
expr ::= id | num
```

```
Stmt stmt() {
   switch (lookahead) {
      case PRINT:
         match(PRINT); match("(");
         Expr e = expr();
         match(")");
         return PrintStmt(e);
      case ID:
         id = lexer_id_val();
         match(ID); match("=");
         Expr e = expr();
         return AssignStmt(id, e);
      default:
          error("syntax error");
   }
}
```

Here the function stmt() returns a Stmt object, and function expr() returns an Expr object.



inheritance hierarchy

Compiling vs. interpreting

Compiling: our language processor will generate code

Add an 'emit()' method to each custom class that will generate code for that class.

Details will depend on the output language.

```
class PrintStmt {
   Expr e;
   …
   // generate code
   void emit() {
       print("printf(");
       e.emit()
       print(");");
   }
}

class Expr {
   String id;
   …
   // generate code
   void emit() {
       print(id);
   }
}
```

Compiling vs. interpreting

Interpreting: our language processor will evaluate the input text.

Add an 'eval()' method to each custom class that will "evaluate" an object of that class.

In this example the return types of eval() are different for Expr and Stmt classes.

```
class PrintStmt {
   Expr e;
   …
   // execute the print statement
   void eval(environment) {
       x = e.eval(environment);
       print(x);
   }
}

class Expr {
   String id;
   …
   // evaluate the expression
   Value eval(environment) {
       return environment(id);
   }
}
```

Building big language processors

Industrial-strength compilers are very large and complicated.

Lots of specialized expertise is need to build them.

For bigger problems, you will probably want to use a "compiler-compiler" like Bison or JavaCC:

- These tools compile grammar definitions into code
- You extend the pure grammar definitions with "action code" that is run when a production is "reduced".

An example Bison grammar rule

```
expseq1: exp
       | expseq1 ',' exp ;
```

Summary

- To do more than parsing, create a syntax tree during parsing
- Compiler: provide the methods with an emit() function
- Interpreter: provide the methods with an eval() function
- To build larger language processors, use specialized tools

Read the "Dragon" book for more on compiling. (Aho, Sethi, Ullman)