Parsing expression grammar

Zaki Mughal

University of Houston:
CougarCS

2013 Mar 28

Parsing expression grammars can be used to match text.

Parsing expression grammars can be used to match text.

So can **regular expressions**.

Parsing expression grammars can be used to match text.

So can **regular expressions**.

Why learn both?

Regular expressions can't match:

Regular expressions can't match:

1 + 2 * (1 + 4)

Regular expressions can't match:

$1 + 2 * (1 + 4)$

```
<html>
  <title> HTML </title>
  <body>
    is also <b> nested </b>
  </body>
</html>
```

Regular expressions can't match:

1 + 2 * (1 + 4)

```
<html>
  <title> HTML </title>
  <body>
    is also <b> nested </b>
  </body>
</html>
```

```
#include<string.h>
int main(int argc, char** argv) {
  if ( argc >= 3
      && strcmp(argv[1], argv[2]) == 0 ) {
    return 1;
  }
  return 0;
}
```

We need a grammar.

Grammars are recursive.

Here's part of a grammar:

```
ClassBody:
    { { ClassBodyDeclaration } }

ClassBodyDeclaration:
    ;
    {Modifier} MemberDecl
    [static] Block

MemberDecl:
    MethodOrFieldDecl
    void Identifier VoidMethodDeclaratorRest
    Identifier ConstructorDeclaratorRest
    GenericMethodOrConstructorDecl
    ClassDeclaration
    InterfaceDeclaration

Block:
    { BlockStatements }

BlockStatements:
    { BlockStatement }

BlockStatement:
    LocalVariableDeclarationStatement
    ClassOrInterfaceDeclaration
    [Identifier :] Statement
```

This is from the Java Language Specification

A single production rule:

```
ClassBodyDeclaration:
```

(non-terminal)

A single production rule:

```
ClassBodyDeclaration:            (non-terminal)
    ;                            (alternative 1)
```

A single production rule:

```
ClassBodyDeclaration:           (non-terminal)
    ;                           (alternative 1)

    {Modifier} MemberDecl       (alternative 2)
```

A single production rule:

```
ClassBodyDeclaration:          (non-terminal)
    ;                          (alternative 1)

    {Modifier} MemberDecl      (alternative 2)



    [static] Block             (alternative 3)


```

A single production rule:

```
ClassBodyDeclaration:       non-terminal matches:
    ;                       matches a literal ; or

    {Modifier} MemberDecl   0 or more Modifier non-terminal
                            followed by a MemberDecl
                            non-terminal or

    [static] Block          optional (0 or 1) terminal
                            (the token static)
                            followed by a Block non-terminal
```

But there's a problem

But there's a problem
when you have alternatives

But there's a problem

when you have alternatives
you have ambiguity

But there's a problem

when you have alternatives

you have ambiguity

Which alternative to follow — multiple trees

Let's eat Grandma!

Let's eat Grandma!

uh. . .

Let's eat Grandma!

uh. . .

Let's eat, Grandma!

Let's eat Grandma!

uh. . .

Let's eat, Grandma!

wipes brow

- Ambiguity happens often with (context-free) grammars[1].

[1] These are part of the Chomsky hierarchy along with regular expressions.

- Ambiguity happens often with (context-free) grammars[1].
- Alternatives: leftmost, rightmost

[1] These are part of the Chomsky hierarchy along with regular expressions.

- PEG is simpler.

- PEG is simpler.
- It follows the first alternative that matches.

Let's write a calculator!

- Operation: +- */ (with precedence)
- parentheses group operations
- variable assignment (a – z)

Future:

- longer variable names
- negative sign (-1)
- decimal numbers (1.25)
- functions (ln), constants (π)
- implicit multiplication ($2a$)

- The Packrat Parsing and Parsing Expression Grammars Page
- peg/leg (C)
- pyparsing (Python)
- Pegex (Perl)