Habanero Extreme Scale Software Research Project
Comp215: Parsing

Zoran Budimlić (Rice University)
“It depends on what the meaning of the word 'is' is.”

- Bill Clinton
A derivation tree is the tree resulting from applying productions to rewrite start symbol

A parse tree is the same tree starting with terminals and building back to the start symbol

A parse tree is a useful data structure to reason about the meaning of the program/data

This is what we will be constructing today
Parsing

The process of determining whether a given sentence belongs to the language defined by the grammar

Can the start symbol derive the given sentence?

If successful, the sentence is a valid sentence

It would be useful to produce a data structure that represents corresponds to what the sentence ‘means’

If failed, the sentence is invalid

If the parsing fails, it would be nice to produce a useful error message that explains why it failed

Yes, we are talking to you, IntelliJ people!

Two approaches in general

Expanding from the start symbol to the whole sentence (top down)
Reduction from the whole sentence to start symbol (bottom up).
Parsing methods

Universal:

There exist algorithms that can parse any context free grammar. These algorithms are too inefficient to be used anywhere. What is considered efficient? Scan the program (from left to right) once.

Top-down parsing

Build the parse tree from root to leave (using leftmost derivation, why?)

Recursive descent and LL parser

Bottom-up parsing

Build the parse tree from leaves to root

Operator precedence parsing, LR (SLR, canonical LR, LALR)
Recursive Descent

Recursive descent parsing associates a function with each nonterminal in the grammar (you can make functions for terminals as well if you wish)

```java
EvilLaugh parse(tokenList){
    if (tokenList.head() == "mwa"){
        cackle = parse(tokenList.tail());
        return new EvilLaugh("mwa", cackle);
    }
}
```
Multiple productions

When there is more than one production, you have to decide which one to use

```
<EvilLaugh> ::= mwa <EvilCackle>
<EvilCackle> ::= ha <EvilCackle>
<EvilCackle> ::= ha!
```

EvilCackle parse(tokenList) {
    if (tokenList.head() == "ha") {
        cackle = parse(tokenList.tail());
        return new EvilCackle("ha", cackle);
    } else if (tokenList.head() == "ha!")
        return new EvilCackle("ha!, emptyCackle);
}
Handling errors

If you cannot create the non-terminal, return an error

```
<EvilLaugh> ::= mwa <EvilCackle>
<EvilCackle> ::= ha <EvilCackle>
<EvilCackle> ::= ha!

EvilCackle parse(tokenList){
    if (tokenList.head() == "ha"){
        cackle = parse(tokenList.tail());
        if (cackle == Error) return Error;
        return new EvilCackle("ha", cackle);
    }else if (tokenList.head() == "ha!")
        return new EvilCackle("ha!, emptyCackle);
    else return Error;
}
```
Left recursion

What do we do if a production has a left recursion?

<EvilLaugh> ::= mwa <EvilCackle> !
<EvilCackle> ::= <EvilCackle> ha
<EvilCackle> ::= ha

EvilCackle parse(tokenList){
    cackle = parse(tokenList);
    if (tokenList.last() == “ha”) return new EvilCackle(cackle, “ha”);
}

Oh no! Infinite recursion!
Left recursion

If a production contains a left recursion, that's a **direct left recursion**:

```plaintext
<EvilCackle> ::= <EvilCackle> ha
```

A grammar is said to have **indirect left recursion** if it is possible, starting from any non-terminal of the grammar, to derive a sentenial form beginning with that symbol:

```plaintext
E ::= T + id
T ::= id
T ::= E * id
```
Eliminating left recursion

If you have a left-recursive production and a non-left recursive production:

\[ <A> ::= <A> \text{ after\_recursion} \]
\[ <A> ::= \text{non\_left\_recursive} \]

Replace that with:

\[ <B> ::= \text{after\_recursion} \mid \text{after\_recursion} <B> \]
\[ <A> ::= \text{non\_left\_recursive} <B> \mid \text{non\_left\_recursive} \]

Do this for every left-recursive production and all corresponding non-left-recursive productions
Our example

<EvilLaugh> ::= mwa <EvilCackle> !
<EvilCackle> ::= <EvilCackle> ha
<EvilCackle> ::= ha
Multiple productions

If there are multiple productions whose RHS starts with a non-terminal, you may have to backtrack.

<Expr> ::= <Term> + <Term>
<Expr> ::= <Factor> * <Term>
<Term> ::= id
<Factor> ::= const

Expr parse(tokenList)

    term = Term.parse(tokenList);
    if (term == Error)
    
        factor = Factor.parse(tokenList)
        . . . // consume “*” and parse the Term

    }else{

        . . .// consume “+” and parse the Term

    }
Predictive parser

Recursive descent parser that does not require backtracking

```plaintext
<EvilLaugh> ::= mwa <EvilCackle>
<EvilCackle> ::= ha <EvilCackle>
<EvilCackle> ::= ha!

EvilCackle parse(tokenList) {
    if (tokenList.head() == "ha") {
        cackle = parse(tokenList.tail());
        return new EvilCackle("ha", cackle);
    } else if (tokenList.head() == "ha!") {
        return new EvilCackle("ha!, emptyCackle);
    }
}
```
“There are only two kinds of languages: the ones people complain about and the ones nobody uses.”

- Bjarne Stroustrup
Recursive descent in Functional Java

tokens come in a list - perfect!
recursion - perfect!
backtracking - easy, we never overwrite anything
make a class for every non-terminal and terminal
store the components of the non-terminal inside the class
store the value of the terminal inside the class
implement a build() method
  returns a parsed object and a list of remaining tokens
  use Pair<> to do this
how to handle errors?
  use Optional
Live coding

<Expr> ::= <Term> | <Term> + <Expr>
<Term> ::= id | id * <Term>