COMP30026 Models of Computation

Parsing Techniques

Harald Søndergaard

Lecture 19

Semester 2, 2018

The Parsing Problem

Given a context-free grammar, the task of a parser is to take an input string w and reconstruct the derivation of w (or discover that this is impossible).

Some parsing algorithm are fast, but work only for restricted types of grammars.

Others, such as the CYK algorithm, work for all context-free grammars, but tend to have bad (cubic) time complexity.

Ideally, we would like for the parser to simply traverse the input from left to right, with no backtracking.

Parsing Techniques

Much is known about restricted classes of grammars that allow for fast parsing.

Top-down parsing algorithms build parse trees from the top down (starting at the root of the parse tree, the start symbol).

Bottom-up parsing algorithms build them from the bottom up (starting at the leaves of the parse tree).

Some parsing algorithms build leftmost derivations; others build rightmost derivations.

Parsing Algorithms

Some top-down parsing algorithms:

- LL parsing, for LL grammars.
- Recursive descent, for LL grammars.
- Recursive descent with backtracking, for arbitrary grammars.

Some bottom-up parsing algorithms:

- Shift-reduce parsing, for LR grammars.
- Operator-precedence parsing, for simple operator grammars.
- CYK parsing (named for its designers, Cocke, Younger and Kasami), for Chomsky Normal Form grammars.

Top-Down Parsing

Here is how we can parse the input *xyyz*, using the grammar below.

$$\begin{array}{lcl} S & \rightarrow & x\ A \\ A & \rightarrow & z \\ A & \rightarrow & y\ A \end{array}$$

Want	Still have
S	$xyyz$
xA	$xyyz$
A	yyz
yA	yyz
A	yz
yA	yz
A	z
z	z
ϵ	ϵ

In Lecture 18, we saw how to build a PDA from a CFG—the "Want" column here is the stack of that PDA.

Predictive Parsing

The example worked out nicely, because whenever we met a grammar variable V on the parse stack, there was at most one rule for V that applied.

We only had to look at the next input symbol to decide how to expand V.

Unfortunately not all grammars are that well-behaved, even if they are unambiguous.

Sometimes a recalcitrant grammar can be transformed to an equivalent well-behaved grammar, sometimes it can't.

Problem 1: Left-Recursion

A grammar for generating powers of 10 in decimal notation:

$$S \rightarrow 1 \mid S\ 0$$

Top-down recognition of 100 would have to backtrack, and having more than 1 symbol look-ahead does not help. A naive approach will not terminate.

On the other hand, the grammar

$$\begin{array}{lcl} S & \rightarrow & 1\ T \\ T & \rightarrow & \epsilon \mid 0\ T \end{array}$$

generates the same language and does allow for linear, predictive parsing.

Left-Recursion Removal

$$A \rightarrow A\,\alpha_1 \mid \cdots \mid A\,\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

The first n rules of A are left-recursive. We can eliminate the left-recursion by transforming the grammar to

$$\begin{array}{lcl} A & \rightarrow & \beta_1\ A' \mid \cdots \mid \beta_m\ A' \\ A' & \rightarrow & \alpha_1\ A' \mid \cdots \mid \alpha_n\ A' \mid \epsilon \end{array}$$

For example, the rules $E \rightarrow E + T \mid T$ turn into

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \end{array}$$

Problem 2: Non-Determinism

Below is a grammar for the regular language $x^*(y \cup z)$.

$$\begin{array}{lcl} S & \to & A \\ S & \to & B \\ A & \to & x\ A \\ A & \to & y \\ B & \to & x\ B \\ B & \to & z \end{array}$$

Want	Still have
S	xxz
A	xxz
xA	xxz
A	xz
xA	xz
A	z

$\Leftarrow$ Bad choice

Left Factoring

In some cases, we can eliminate non-determinism in rules.

Given

$$A \quad \rightarrow \quad \alpha\ \beta_1 \mid \cdots \mid \alpha\ \beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$$

(where α is the longest common prefix), form

$$\begin{array}{lcl} A & \rightarrow & \alpha\ A' \mid \gamma_1 \mid \cdots \mid \gamma_m \\ A' & \rightarrow & \beta_1 \mid \cdots \mid \beta_n \end{array}$$

For example, the grammar fragment below, left, becomes the one on the right (both are ambiguous, by the way).

$$S \quad \rightarrow \quad \mathtt{i}\ E\ \mathtt{t}\ S \mid \mathtt{i}\ E\ \mathtt{t}\ S\ \mathtt{e}\ S \mid \mathtt{o} \qquad \begin{array}{lcl} S & \rightarrow & \mathtt{i}\ E\ \mathtt{t}\ S\ S' \mid \mathtt{o} \\ S' & \rightarrow & \epsilon \mid \mathtt{e}\ S \end{array}$$

Problem 3: Epsilon Rules

Below is a grammar for the regular language $x \cup xx$:

$$\begin{array}{lcl} S & \to & A\,x \\ A & \to & x \\ A & \to & \epsilon \end{array}$$

Want	Still have	
S	x	
Ax	x	
xx	x	$\Leftarrow$ Bad choice
x	ϵ	

The presence of ϵs make it harder to determine whether a grammar is amenable to predictive top-down parsing.

LL Grammars

There are, however, ways of automatically determining whether a grammar can be used for predictive top-down parsing.

The set of languages that admit a so-called LL(k) grammar can be parsed predictively using k symbols of look-ahead.

These languages make up the class of LL(k) languages, a proper sub-class of the context-free class.

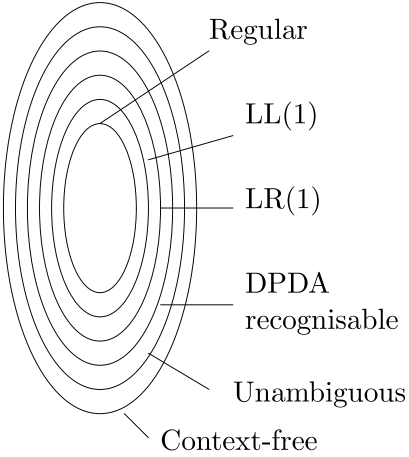
Parsing Programming Languages

Some programming language constructs fall outside of LL(k) for all k.

The designers of Python carefully chose its syntax to be LL(1).

Still, it would be wrong to say that Python, or other common programming languages are contex-free, because there are certain "syntactic" aspects that cannot be captured in a context-free grammar—for example that "a variable must have been declared before its first use."

Important Sub-Classes of CFLs



Recursive Descent

If a language has an LL grammar then there is a simple systematic way of writing a parser for it.

The technique of recursive descent consists of writing a set of (normally mutually recursive) parsing functions, one per grammar variable (or syntactic category).

An example, written in Haskell, is the parser for propositional formulas that you can find in the file `Exp.hs` in Worksheet 2.

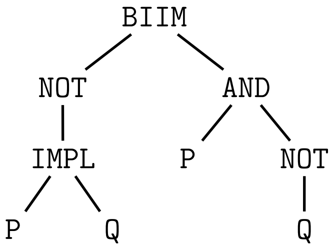
Example: Parsing Propositional Formulas

We want to take a string such as

```
((~(P => Q)) <=> (P /\ (~Q)))
```

and determine whether it is well-formed (syntactically correct).

Moreover, we want to discover its hidden structure:



Example: Parsing Propositional Formulas

Expressed as Haskell comments, the readable grammar for propositional formulas is

```
--   exp -> var
--        |  ( ~ exp )
--        |  ( exp /\ exp )
--        |  ( exp \/ exp )
--        |  ( exp => exp )
--        |  ( exp <=> exp )
--        |  ( exp <+> exp )
--        |  ( exp )
```

However, that is not LL(1)—lots of left factoring is needed.

An LL(1) Grammar for Propositional Formulas

Here is a less readable equivalent LL(1) grammar:

```
--   exp -> var
--        |  ( par )
--   par -> ~ exp
--        |  exp rem
--   rem -> /\ exp
--        |  \/ exp
--        |  => exp
--        |  <=> exp
--        |  <+> exp
--        |  epsilon
```

Recursive Parser Functions

From the last grammar, it becomes quite easy to implement a recursive descent parser.

It will consist of three functions, one for `exp`, one for `par`, and one for `rem`.

The three will call upon each other for help.

Recursive Parser Functions

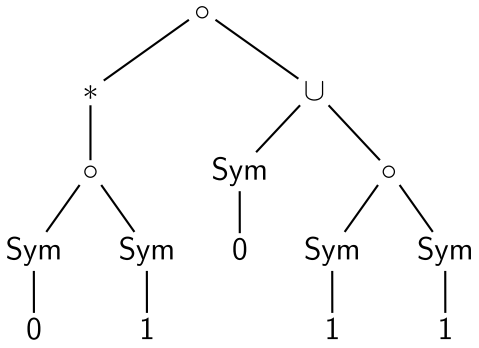
```
function EXP(input)
    if next(input) is an identifier then
        get identifier
    else
        if next(input) is left parenthesis then
            get left parenthesis
            call PAR(input)
            if next(input) is right parenthesis then
                get right parenthesis
            else
                error
        else
            error
```

See Worksheet 2 for how to do this in Haskell.

Recursive Parser Functions

Worksheet 4 asks you to complete a recursive descent parser for regular expressions.

We want to be able to take input such as (01)∗(0+11) and produce a Haskell expression such as



```
Concat (Star (Concat (Sym '0') (Sym '1')))
       (Union (Sym '0') (Concat (Sym '1') (Sym '1')))
```

which captures the structure tree shown on the right.