CA4003 - Compiler Construction

Introduction to Parsers

David Sinclair

Limitations of Regular Expressions

Consider the following set of regular expression:

`digits`	$=$	[0-9]+
`sum`	$=$	`expr` "+" `expr`
`expr`	$=$	"(" `sum` ")" | `digits`

which is trying to recognise an expression such as $(12 + (52 + 7))$ with balanced parentheses.

But regular expressions cannot count.

- An automaton with N states can't handle an expression with more than N sets of balanced parentheses.

Limitations of Regular Expressions [2]

It is even worse!

Substituting `sum` into `expr` yields:
`expr` = "(" `expr` "+" `expr` ")" | `digits`

Substituting `expr` in again yields:
`expr` = "(" "(" `expr` "+" `expr` ")" | `digits` "+" `expr` ")" |
`digits`

and the right hand side now has more `expr` terms.

Adding Recursion

Adding recursion solves the problem.

- actually we will use mutual recursion

We will not need *alternation* except at the top level.
`expr` = `a b`(`c` | `d`)`e`
can be written as

```
aux  = c
aux  = d
expr = a b aux e
```

And Kleen closure is not needed anymore.
`expr` = (`a b c`)*
can be written as

```
expr = (a b c) expr
expr = ε
```

This simplified notation is called a *context free grammar*.

Context Free Grammers (CFGs)

Context free syntax is specified with a context free grammar.
Formally, a `CFG` G is a 4-tuple (V_t, V_n, S, P), where:

V_t is the set of terminal symbols in the grammar.
For our purposes, V_t is the set of tokens returned by the scanner.

V_n are the *nonterminals*, a set of syntactic variables that denote sets of (sub)strings occurring in the language.
These are used to impose a structure on the grammar.

Context Free Grammers (CFGs) [2]

S is a distinguished nonterminal ($S \in V_n$) denoting the entire set of strings in $L(G)$.
This is sometimes called a *goal symbol*.

P is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language.
Each production must have a single non-terminal on its left hand side.

The set $V = V_t \cup V_n$ is called the *vocabulary* of G

Terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V^*$
- $u, v, w, \ldots \in V_t^*$

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \rightarrow \gamma$

Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps.

If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G.

$L(G) = \{w \in V_t^* | S \Rightarrow^+ w\}, w \in L(G)$ is called a *sentence* of G.

Note, $L(G) = \{\beta \in V^* | S \Rightarrow^* \beta\} \cap V_t^*$

Notation - BNF

Grammars are often written in Backus-Naur form (BNF).
Example:

1	`<goal>`	::=	`<expr>`
2	`<expr>`	::=	`<expr><op><expr>`
3		|	`num`
4		|	`id`
5	`<op>`	::=	$+$
6	`<op>`	::=	$-$
7	`<op>`	::=	$*$
8	`<op>`	::=	$/$

This describes simple expressions over numbers and identifiers.
In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with `typewriter` font or <u>underline</u>
3. productions as in the example

Derivations

Consider the sentence $\texttt{x} + \texttt{2*y}$. Using the CFG from the previous slide we could get the following series of *derivations*.

```
<goal>  ⇒  <expr>
        ⇒  <expr><op><expr>
        ⇒  <expr><op><expr><op><expr>
        ⇒  <id,x><op><expr><op><expr>
        ⇒  <id,x> + <expr><op><expr>
        ⇒  <id,x> + <num,2><op><expr>
        ⇒  <id,x> + <num,2>*<expr>
        ⇒  <id,x> + <num,2>*<id,y>
```

So $\texttt{<goal>} \Rightarrow^* \texttt{id} + \texttt{num*id}$.
A sequence of production applications is a *derivation* or a *parse*.
Parsing is the process of discovering a derivation for a statement.

Derivations [2]

At each step, we chose a non-terminal to replace.

This choice can lead to different derivations.

Two are of particular interest:

- leftmost derivation
  - The leftmost non-terminal is replaced at each step.
- rightmost derivation
  - The rightmost non-terminal is replaced at each step .

The previous example was a leftmost derivation.

Rightmost Derivation

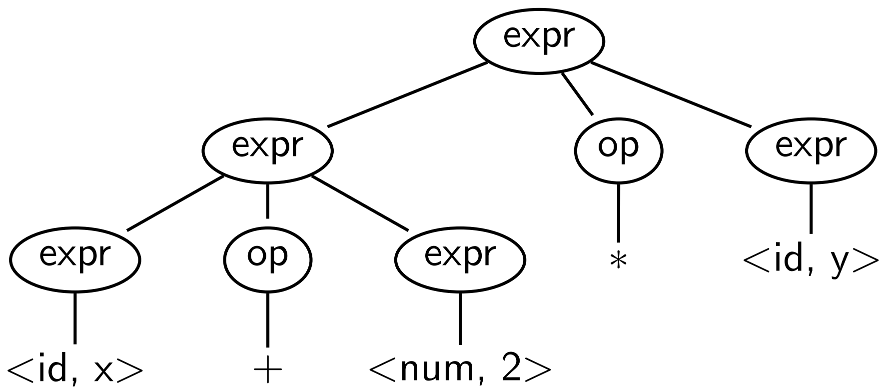
Taking the same statement and performing a rightmost derivations would yield:

<goal> ⇒ <expr>
⇒ <expr><op><expr>
⇒ <expr><op><id,y>
⇒ <expr>*<id,y>
⇒ <expr><op><expr>*<id,y>
⇒ <expr><op><num,2>*<id,y>
⇒ <expr> + <num,2>*<id,y>
⇒ <id,x> + <num,2>*<id,y>

Again, <goal> $\Rightarrow^*$ id + num*id, but the structure induced is different.

Rightmost Derivation [2]

The parse tree for the rightmost derivation is:



If we evaluate this tree we would get the “wrong” answer, $(x + 2) * y$, when what was intended was $x + (2 * y)$.

Precedence

The problem with the grammar is that there evaluation order implied by the grammar.
One way to add *precedence* to the grammar is to add additional structure to the grammar.

1	<goal>	::=	<expr>
2	<expr>	::=	<expr> + <term>
3		|	<expr> - <term>
4		|	<term>
5	<term>	::=	<term> * <factor>
6		|	<term> / <factor>
7		|	<factor>
8	<factor>	::=	num
9		|	id

Precedence [2]

`terms` **must** be derived from `expr`, *expressions*.

`factors` **must** be derived from `terms`.

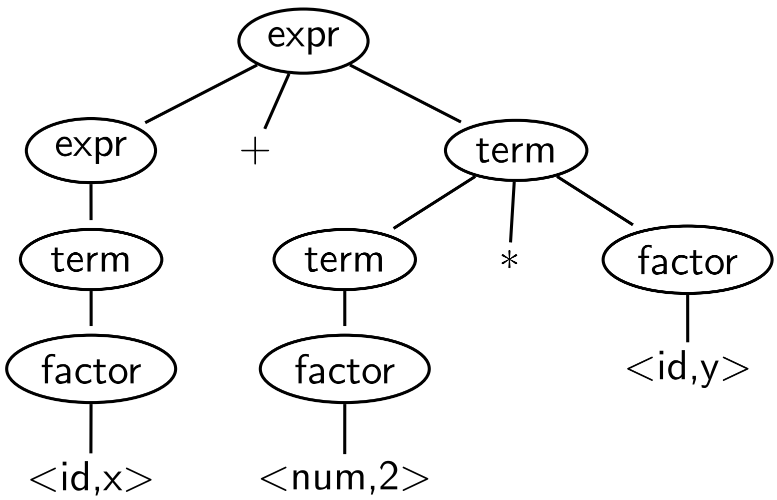
By adding this additional structure we imply an evaluation order and we get the “correct” tree.

Precedence [3]

```
<goal>  ⇒  <expr>
        ⇒  <expr> + <term>
        ⇒  <expr> + <term> * <factor>
        ⇒  <expr> + <term> * <id,y>
        ⇒  <expr> + <factor> * <id,y>
        ⇒  <expr> + <num,2> * <id,y>
        ⇒  <term> + <num,2> * <id,y>
        ⇒  <factor> + <num,2> * <id,y>
        ⇒  <id,x> + <num,2> * <id,y>
```

<goal>$\Rightarrow^*$ id + num * id, as before, but this time the parse tree is:

Precedence [4]



Evaluating the tree, *treewalk evaluation*, results in $x + (2 * y)$.

Another approach is to add precedence and associativity information to tokens.

Ambiguity

A grammar is *ambiguous* if a sentence admits two or more derivations.
Consider the following grammar:

```
<stmt>   ::=  if <expr> then <stmt>
           |  if <expr> then <stmt> else <stmt>
           |  other stmts
```

The sentence

$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_1 \texttt{ else } S_2$$

has two derivations.

Can you find them?

Ambiguity [2]

Rearranging the grammar eliminates the ambiguity. This following grammar generates the same language, but with the rule:

match each `else` *with the closest unmatched* `then`

```
<stmt>      ::= <matched>
              | <unmatched>
<matched>   ::= if <expr> then <matched> else <matched>
              | other stmts
<unmatched> ::= if <expr> then <stmt>
              | if <expr> then <matched> else <unmatched>
```

This is most likely what the language designer's intended.

The ambiguity generated by the first `if...then..else` grammar is an example of *context free ambiguity*.

Ambiguity [3]

In addition to context free ambiguity, ambiguity can be context sensitive. Context sensitive confusions can arise from *overloading*.

For example:
`a = f(17)`
In many Algol-like languages, `f` could be a function or a subscripted variable.

Disambiguating this statement requires context:

- need *values* of declarations, and
- really an issue of *type*.

Rather than complicate parsing, this type of ambiguity is handled separately.