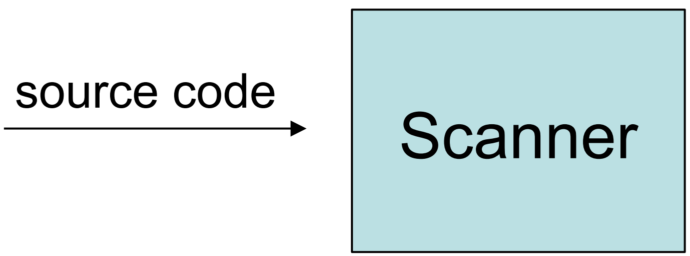
Compilers:
Parsing & Context-Free Grammars

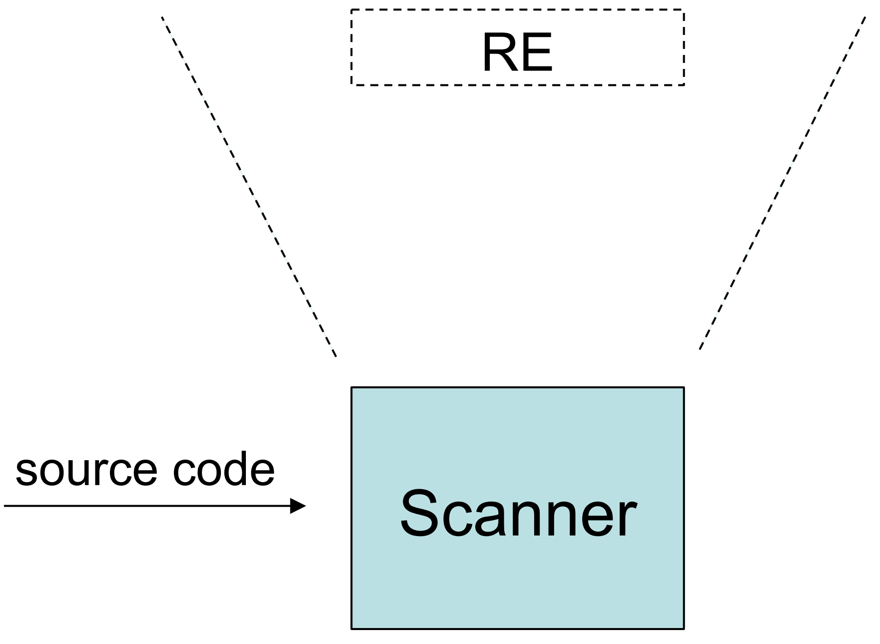
Dr. Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

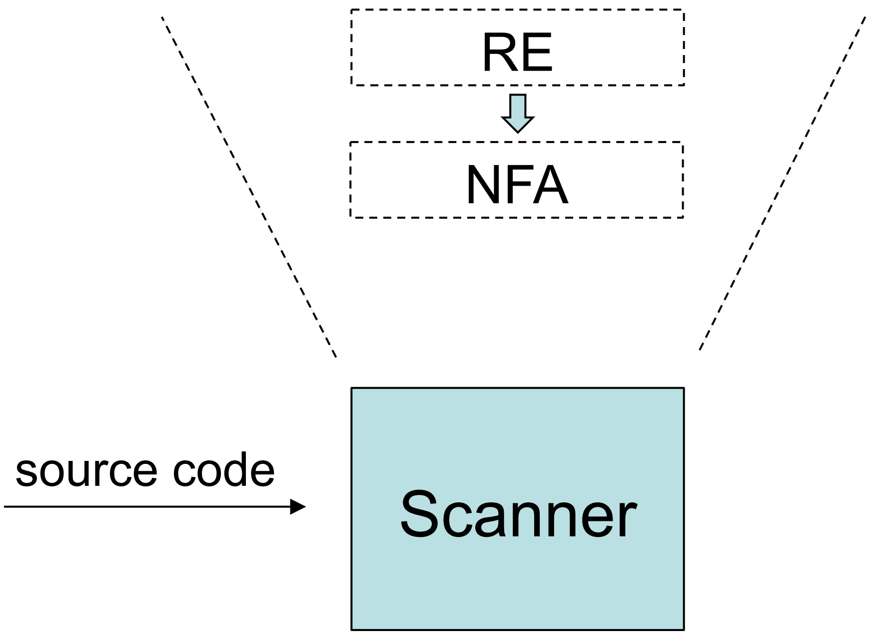
Where we are…

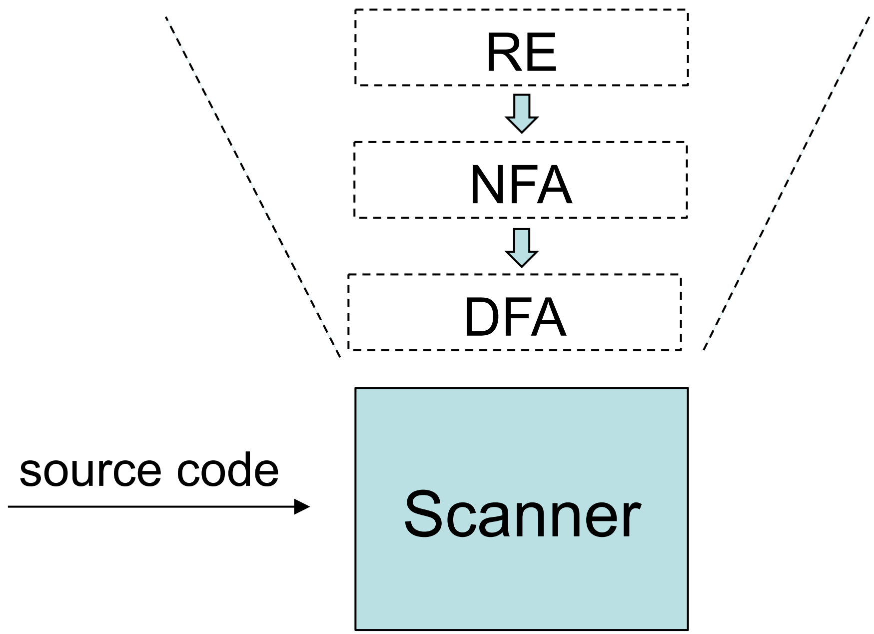
- ~~Admin and overview~~
- ~~Lexical analysis~~
- **Parsing**
- Semantic analysis
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review

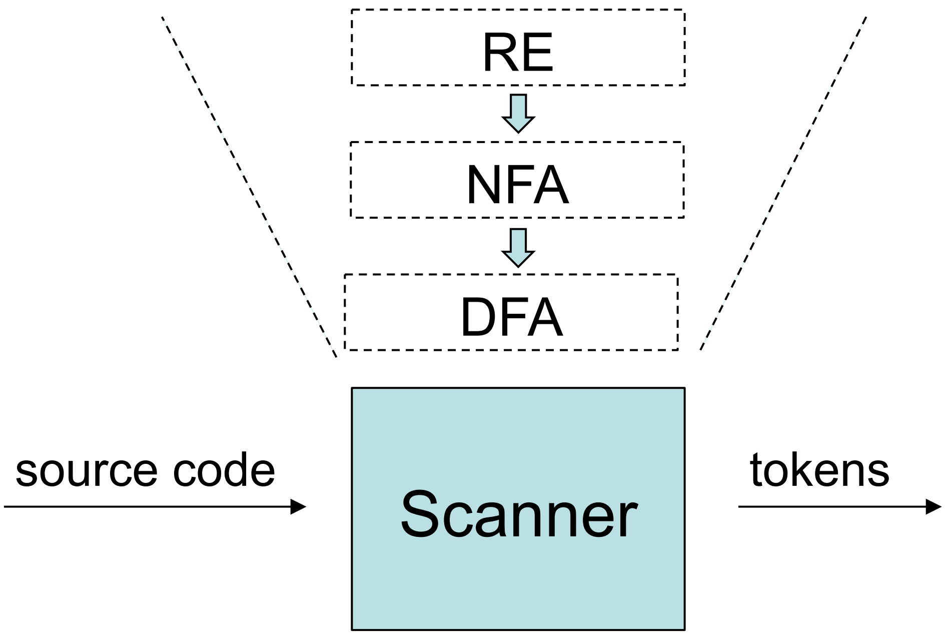
source code

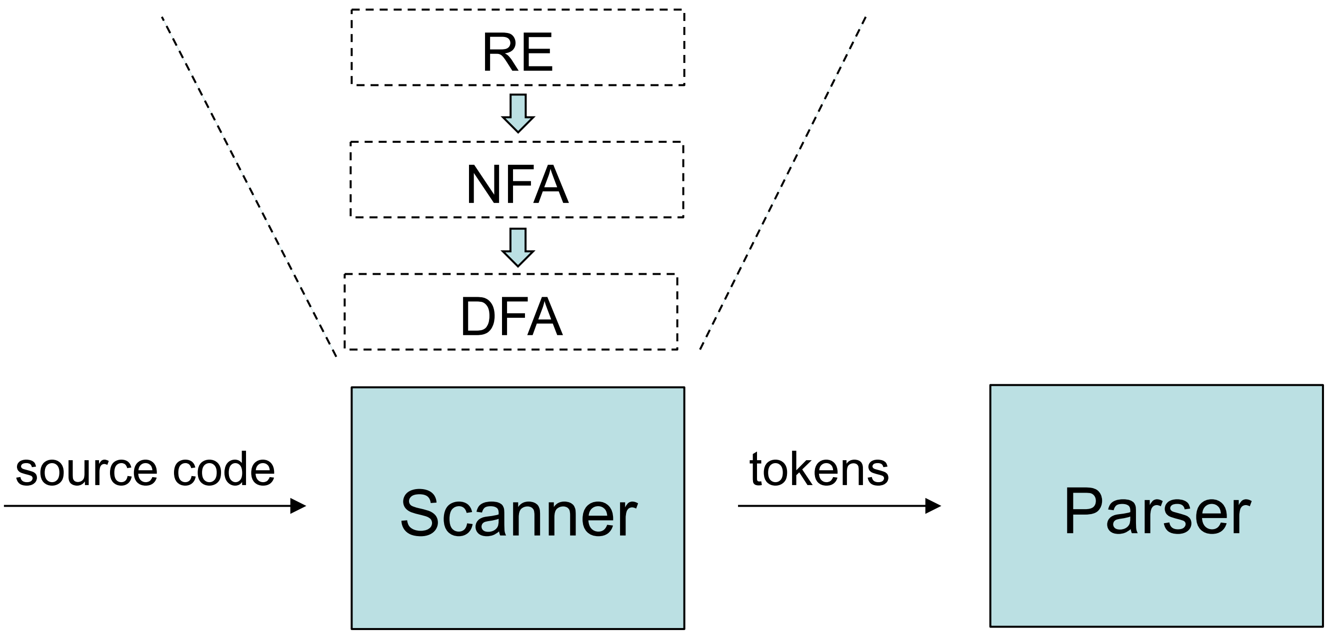

source code
Scanner

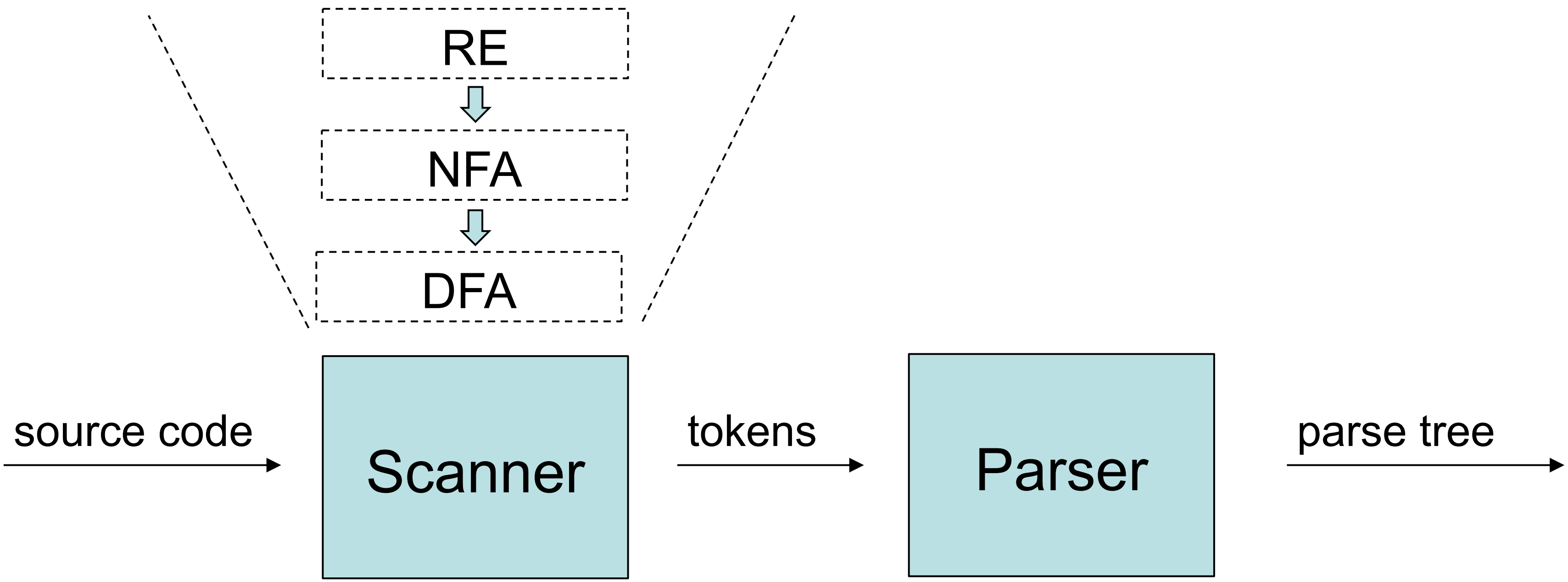

RE
source code
Scanner


RE
NFA
source code
Scanner


RE
NFA
DFA
source code
Scanner


RE
NFA
DFA
source code
Scanner
tokens


RE
NFA
DFA
source code
Scanner
tokens
Parser


RE
NFA
DFA
source code
Scanner
tokens
Parser
parse tree

Objectives

- To introduce *parsing*
- Discuss error recovery in parsing
- Motivate the use of context-free grammars (CFGs)
- Define context-free grammars
- Construct parse trees for context-free grammars
- Introduce ambiguity

Parsing

- Checks the stream of tokens (produced by the scanner) for grammatical correctness.
- Determines if the input is *syntactically well formed.*
  - *OK, we have valid tokens, but have they been put together in a sensible way?*
- Builds an intermediate representation (IR) of the code.

Types of Programming Errors

- Lexical errors: misspellings of keywords, identifiers and operators
- Syntactic errors: misplaced semi-colons, missing or extra braces, **case** without a **switch**, etc
- Semantic errors: such as a function/method of type **void** trying to return a value
- Logical errors: from incorrect reasoning by the programmer, through to statements like

```
if (x = a)
```

Goals of the Syntactic Error Handler

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs

Error Recovery Strategies

- Panic-mode recovery
- Phrase-level recovery
- Error productions
- Global correction

Representing Syntax

- Regular expressions are a powerful tool for specifying allowed tokens, but not so good for syntax
  - Could you write a RE that accepted expressions of this form:

    a * (b + c) ???

- Instead, we use *context-free grammars* (CFGs), or just *grammars* for short

Decaf grammar fragment

<table>
<tr><td>⟨program⟩</td><td>→</td><td>class Program '{' ⟨field_decl⟩* ⟨method_decl⟩* '}'</td></tr>
<tr><td>⟨field_decl⟩</td><td>→</td><td>⟨type⟩ {⟨id⟩ | ⟨id⟩ '[' ⟨int_literal⟩ ']'}$^{+}$, ;</td></tr>
<tr><td>⟨method_decl⟩</td><td>→</td><td>{⟨type⟩ | void} ⟨id⟩ ([{⟨type⟩ ⟨id⟩}$^{+}$,]) ⟨block⟩</td></tr>
<tr><td>⟨block⟩</td><td>→</td><td>'{' ⟨var_decl⟩* ⟨statement⟩* '}'</td></tr>
<tr><td>⟨var_decl⟩</td><td>→</td><td>⟨type⟩ ⟨id⟩$^{+}$, ;</td></tr>
<tr><td>⟨type⟩</td><td>→</td><td>int | boolean</td></tr>
<tr><td>⟨statement⟩</td><td>→</td><td>⟨location⟩ ⟨assign_op⟩ ⟨expr⟩ ;</td></tr>
<tr><td></td><td>|</td><td>⟨method call⟩ ;</td></tr>
<tr><td></td><td>|</td><td>if (⟨expr⟩) ⟨block⟩ [else ⟨block⟩]</td></tr>
<tr><td></td><td>|</td><td>for ⟨id⟩ = ⟨expr⟩ , ⟨expr⟩ ⟨block⟩</td></tr>
<tr><td></td><td>|</td><td>return [⟨expr⟩] ;</td></tr>
<tr><td></td><td>|</td><td>break ;</td></tr>
<tr><td></td><td>|</td><td>continue ;</td></tr>
<tr><td></td><td>|</td><td>⟨block⟩</td></tr>
</table>

Context-Free Grammars

- Consist of four parts:
  - A finite set *N* of symbols called the *nonterminal alphabet*
  - A finite set *T* of symbols called the *terminal alphabet*
  - A finite set of *productions*
  - A *start symbol* from the set *N*
- *N* and *T* should not share any elements

Example

- Nonterminal alphabet $N = \{S, B, C\}$
- Terminal alphabet $T = \{b, c\}$
- Productions:
    1. $S \rightarrow BC$
    2. $B \rightarrow bB$
    3. $B \rightarrow \lambda$
    4. $C \rightarrow ccc$
- Start symbol is S

String Derivation & Languages

1. $S \rightarrow BC$
2. $B \rightarrow bB$
3. $B \rightarrow \lambda$
4. $C \rightarrow ccc$

- *String derivation* is the process applying productions to generate strings
  - For example, *S* can derive **ccc** thus:

$$\underset{1}{S} \Rightarrow \underset{3}{BC} \Rightarrow \underset{4}{C} \Rightarrow ccc \qquad (S \overset{*}{\Rightarrow} ccc)$$

- A grammar can derive many different terminal strings; L(G) is the language defined by a context-free grammar $\mathrm{L(G)} = \{ x : \mathrm{S} \Rightarrow x \text{ and } x \in T\}$

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Another Example

1	*Expr*	$\rightarrow$	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	$\rightarrow$	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	**Sentential Form**

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	Sentential Form
	Expr

Another Example

1	*Expr*	$\rightarrow$	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	$\rightarrow$	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	Sentential Form
	Expr
2	*Expr Op* name

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	**Sentential Form**
	Expr
2	*Expr Op* name
6	*Expr* × name

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	**Sentential Form**
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	**Sentential Form**
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

One way of parsing it:

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name

Another Example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Consider the sentence

(a + b) × c

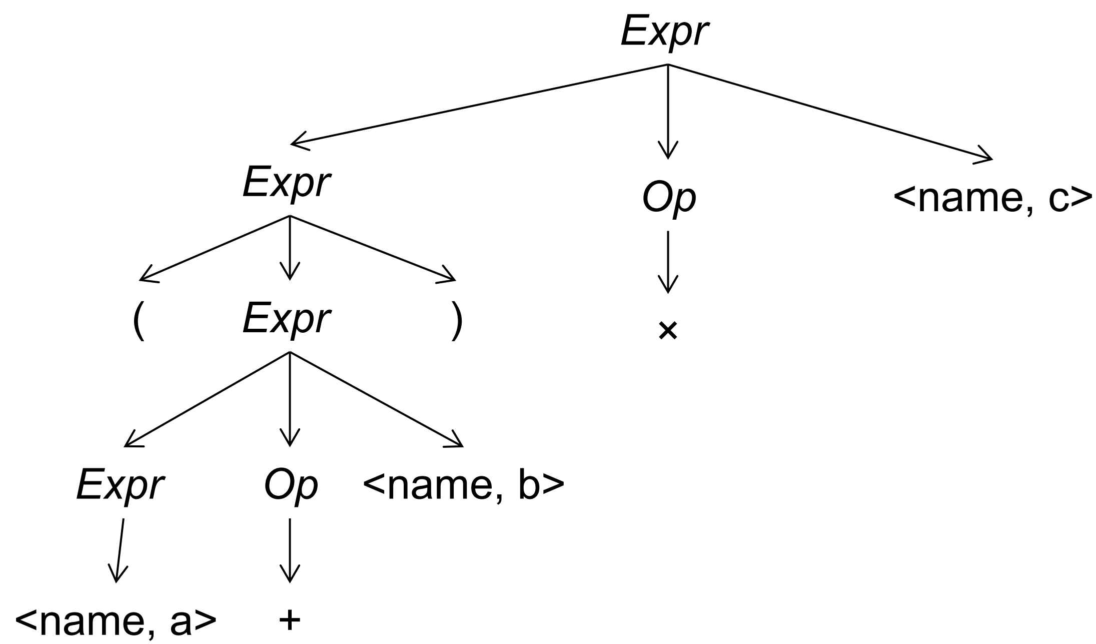
One way of parsing it:

Rule	**Sentential Form**
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

Rightmost Derivation

(a + b) × c

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name



Another alternative

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

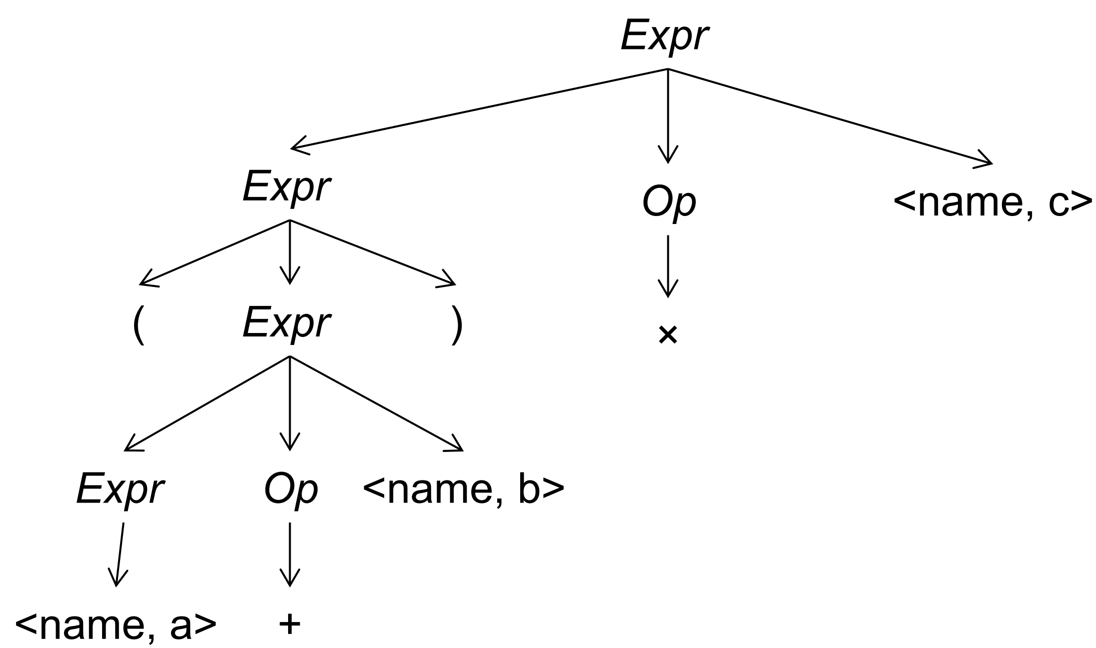
(a + b) × c

Rule	**Sentential Form**
	Expr
2	*Expr Op* name
1	(*Expr*) *Op* name
2	(*Expr Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name

Leftmost Derivation

(a + b) × c

Rule	Sentential Form
	Expr
2	*Expr Op* name
1	(*Expr*) *Op* name
2	(*Expr Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name



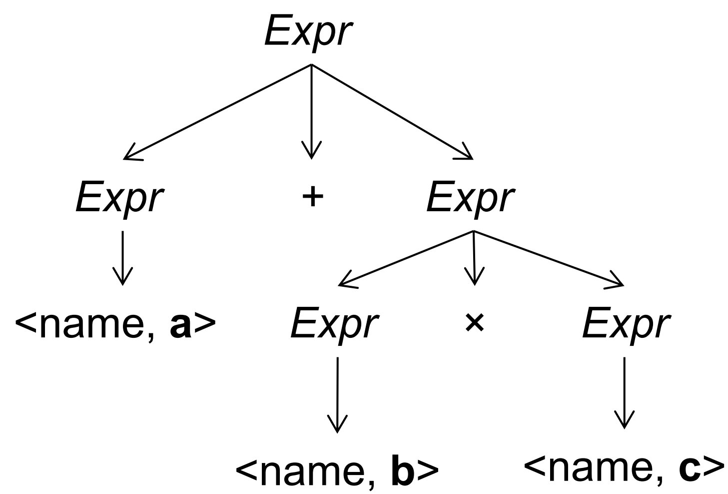
More than One Parse Tree?

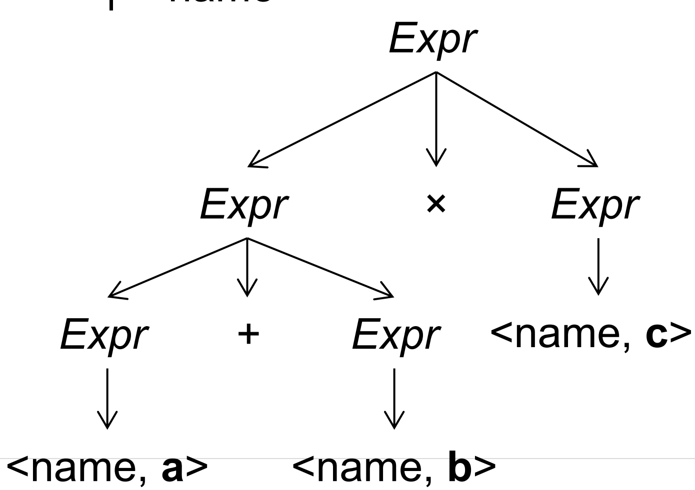
- Some grammars can be awkward – it can be possible to generate more than one parse tree for a given sentence.
- Such a grammar is said to be *ambiguous*

An Ambiguous Grammar

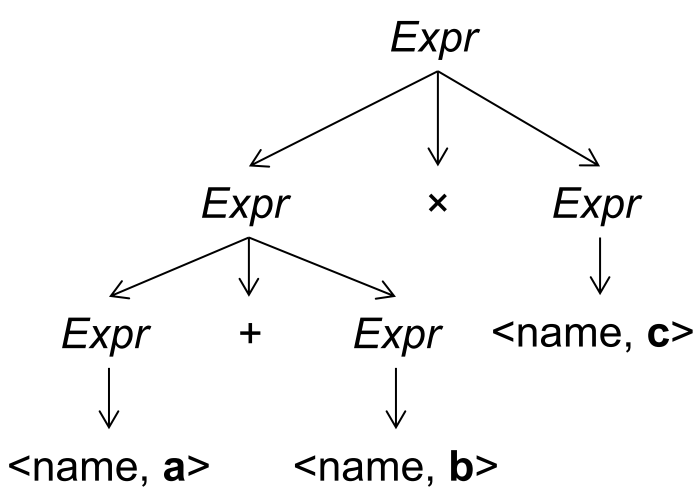
a + b × c

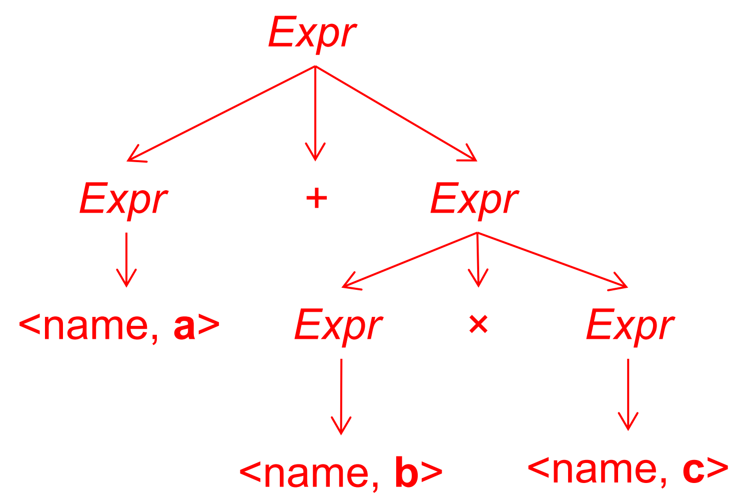
1	*Expr*	→	*Expr* + *Expr*
2		|	*Expr* × *Expr*
3		|	(*Expr*)
4		|	name





Structure Implies Meaning





Implies
(a + b) × c

Implies
a + (b × c)

Removing Ambiguity

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Term*
4	*Term*	→	Term × Factor
5		|	*Factor*
6	*Factor*	→	(*Expr*)
7		|	name

- Now only one possible way to interpret a + b × c
- Ambiguity is removed

  AND
- Ensure expected operator precedence

Left- and Right-Recursion

- A grammar that has rules of the form

  $$A \rightarrow Aa$$

  is said to be left-recursive.
- A grammar with rules of the form

  $$A \rightarrow aA$$

  is said to be right-recursive.
- Sometimes it is useful to eliminate left-recursion, e.g.

  $$A \rightarrow Aa \mid b$$

  can be replaced by the pair of rules

  $$A \rightarrow bA' \text{ and } A' \rightarrow aA' \mid \varepsilon$$

More about this next week!

Summary

- Parsing is…
- A context-free grammar is…
- CFGs used in place of REs because…
- How do you construct a parse tree?
- What is ambiguity? Why is it a problem? What can be done about it?
- What are left- and right-recursive grammars? How can you re-write a grammar to remove left-recursion?

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- **Parsing**
- Semantic analysis
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review