The University for
world-class professionals

Manchester
Metropolitan
University

Compilers: Parsing

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

Objectives

- Describe the principles of top-down and bottom-up parsing
- Examine different ways of implementing both types of parsing
- Describe problems associated with particular implementations of these algorithms

Top Down Parsing

"Top-down parsers begin with the root and grow the tree toward the leaves. At each step, a top-down parser selects a node for some nonterminal on the lower fringe of the tree and extends it with a subtree that represents the right-hand side of a production that rewrites the nonterminal."

(*Engineering a Compiler, p. 96)*

Bottom-up Parsers

"Bottom-up parsers begin with the leaves and grow the tree toward the root. At each step, a bottom-up parser identifies a contiguous substring of the parse tree's upper fringe that matches the right-hand side of some production; it then builds a node for the rule's left-hand side and connects it into the tree."

(*Engineering a Compiler, p. 96)*

Top down parsing

- Top-down parsing corresponds to the way we built parse trees last week:
  - start at the most abstract, gradually expand the tree

Recall from last week

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

- Notice that when parsing an input like “a – b + c”, you start by matching rule 1. Then you have a choice of three rules to expand *Expr*. Which do you choose?

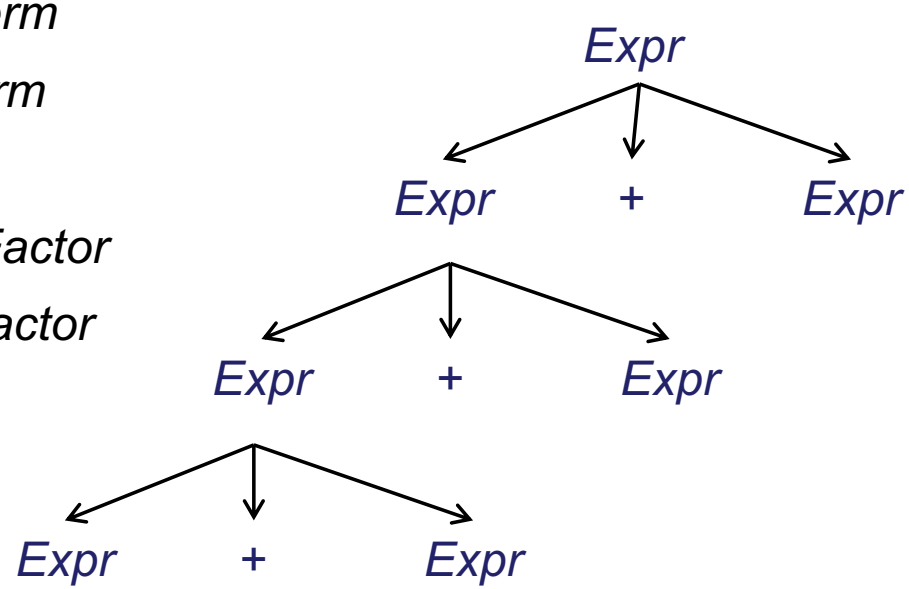
What do we do?

- Which production to use for each nonterminal?
- One Approach: **Backtracking**
  - Treat it as a search problem
  - At each choice point, try next alternative
  - If it is clear that current try fails, go back to previous choice and try something different
- General technique for searching
- Used a lot in classical AI and natural language processing (parsing, speech recognition)

Ok, but…

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

a - b + c



Left-recursion – get rid of it!

- Recall:

  $A \rightarrow Aa \mid b$

Left-recursion – get rid of it!

- Recall:

  $A \rightarrow Aa \mid b$

  can be replaced by the pair of rules

  $A \rightarrow bA'$ and

  $A' \rightarrow aA' \mid \varepsilon$

Left-recursion – example

$A \rightarrow Aa \mid b$

Expr → *Expr* name | *Term*

$A \rightarrow bA'$

$A' \rightarrow aA' \mid \varepsilon$

Left-recursion – example

$A \rightarrow Aa \mid b$

$$\overbrace{\textit{Expr}}^{A} \rightarrow \overbrace{\textit{Expr}}^{A}\ \overbrace{\text{name}}^{a} \mid \overbrace{\textit{Term}}^{b}$$

$A \rightarrow bA'$

$A' \rightarrow aA' \mid \varepsilon$

Left-recursion – example

$A \rightarrow Aa \mid b$

$\overbrace{Expr}^{A} \rightarrow \overbrace{Expr}^{A}\ \overbrace{\text{name}}^{a} \mid \overbrace{Term}^{b}$

$A \rightarrow bA'$

$Expr \rightarrow Term\ Expr'$

$A' \rightarrow aA' \mid \varepsilon$

Left-recursion – example

$A \rightarrow Aa \mid b$

$\overbrace{Expr}^{A} \rightarrow \overbrace{Expr}^{A}\ \overbrace{\text{name}}^{a} \mid \overbrace{Term}^{b}$

$A \rightarrow bA'$

$Expr \rightarrow Term\ Expr'$

$A' \rightarrow aA' \mid \varepsilon$

$Expr' \rightarrow \text{name}\ Expr' \mid \varepsilon$

A → Aa | b

A→ bA' and A' → aA' | ε

So…

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

1	*Goal*	→	*Expr*

A → Aa | b

A→ bA' and A' → aA' | ε

So…

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

1	*Goal*	→	*Expr*

A → Aa | b

A→ bA' and A' → aA' | ε

So…

A A a

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term* b
5	*Term*	→	*Term* × *Factor*
6		|	*Term* ÷ *Factor*
7		|	*Factor*
8	*Factor*	→	(*Expr*)
9		|	name

1 *Goal* → *Expr*

A → Aa | b

A→ bA' and A' → aA' | ε

So…

A A a

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term* b
5	*Term*	→	*Term* × *Factor*
6		|	*Term* ÷ *Factor*
7		|	*Factor*
8	*Factor*	→	(*Expr*)
9		|	name

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*

A → bA'

A → Aa | b

A→ bA' and A' → aA' | ε

So…

A

a

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

b

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
		|	ε

A' → aA' | ε

A → Aa | b

A→ bA' and A' → aA' | ε

So…

A

a

b

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

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	*+ Term Expr'*
		|	ε

A → Aa | b

A→ bA' and A' → aA' | ε

So…

A

a

b

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

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	*+ Term Expr'*
4		|	*- Term Expr'*
5		|	ε

A → Aa | b

A→ bA' and A' → aA' | ε

So…

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

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε

A → Aa | b

A→ bA' and A' → aA' | ε

So…

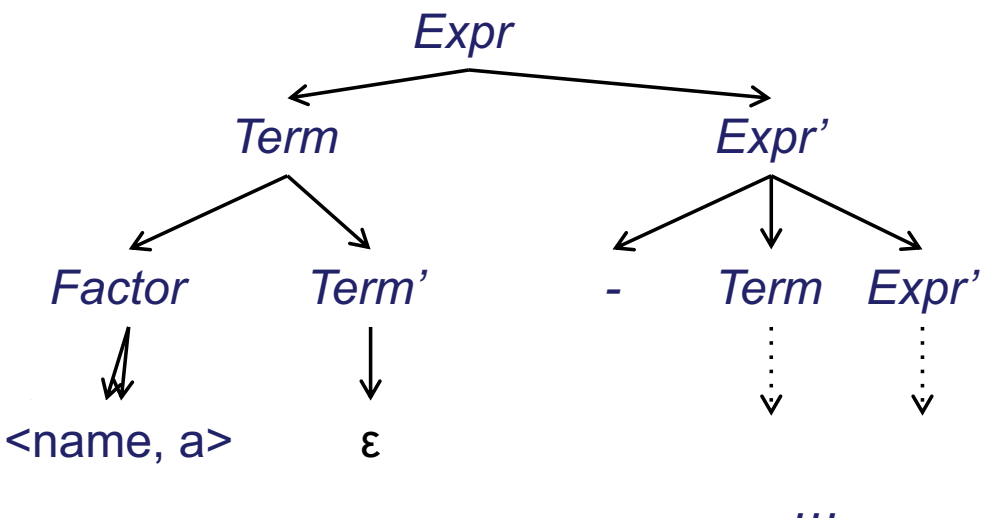
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* × *Factor*
6		|	*Term* ÷ *Factor*
7		|	*Factor*
8	*Factor*	→	(*Expr*)
9		|	name

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	→	× *Factor Term'*
8		|	÷ *Factor Term'*
9		|	ε
10	*Factor*	→	(*Expr*)
11		|	name

So Let's Make Sure…

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	|	× *Factor Term'*
8		|	÷ *Factor Term'*
9		|	ε
10	*Factor*	→	(*Expr*)
11		|	name

a - b + c



Indirect left-recursion

$X \rightarrow Yx$

$Y \rightarrow Zy \mid x$

$Z \rightarrow Xy \mid z$

Indirect left-recursion

$X \rightarrow Yx$

$Y \rightarrow Zy \mid x$

$Z \rightarrow Xy \mid z$

$\Rightarrow$

$X \rightarrow Zyx \mid xx$

$Z \rightarrow Xy \mid z$

Indirect left-recursion

$X \rightarrow Yx$

$Y \rightarrow Zy \mid x$

$Z \rightarrow Xy \mid z$

$\Rightarrow$

$X \rightarrow Zyx \mid xx$

$Z \rightarrow Xy \mid z$

$\Rightarrow$

$X \rightarrow Xyyx \mid zyx \mid xx$

Indirect left-recursion

$X \rightarrow Yx$

$Y \rightarrow Zy \mid x$

$Z \rightarrow Xy \mid z$

$X \rightarrow Zyx \mid xx$

$Z \rightarrow Xy \mid z$

$X \rightarrow Xyyx \mid zyx \mid xx$

*see ‘Test Yourself‘ sheet in Week 5 Moodle.

Backtracking…

- It's costly: can process substrings multiple times.
- On the other hand, in practice it actually doesn't arise all that often.
- An alternative is *lookahead* or *predictive* parsing
  - When there are multiple possible alternatives, look ahead at next character(s) to disambiguate.

First and Follow

- Key concepts used in lookahead parsing

First:

  – For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the start of a sentence derived from α.

Follow:

  For a nonterminal α, FOLLOW(α) contains the set of words that can occur immediately after α in a sentence.

For Our Expression Grammar…

- First(*Factor*) = First(*Term*) = First(*Expr*) = { (, name }
- First(*Expr'*) = { + , - , ε }
- First(*Term'*) = { × , ÷, ε }

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	→	× *Factor Term'*
8		|	÷ *Factor Term'*
9		|	ε
10	*Factor*	→	(*Expr*)
11		|	name

And the Follow Sets…

- Follow(*Expr*) = Follow(*Expr'*) = {) , $ }
- Follow(*Term*) = Follow(*Term'*) = { + , - ,) , $ }
- Follow(*Factor*) = {× , ÷, +, -,) , $ }

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	→	× *Factor Term'*
8		|	÷ *Factor Term'*
9		|	ε
10	*Factor*	→	(*Expr*)
11		|	name

Bottom up parsing

- No need to worry about left-recursive grammars
- Still need to worry about ambiguous grammars
- Bottom-up parsing *reduces* a string to the start symbol by inverting productions.
- example: **int * int + int**

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

productions

Manchester Metropolitan University

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	E

reductions

Implementing bottom up parsers

- Shift
- Reduce
- Bottom-up parsing traces a rightmost derivation in reverse
- Can be implemented using a stack
- Always pick the correct reduction by using handles (reductions that allow further reductions)

Manchester Metropolitan University

int * int + int

T → int

T → int * T

T → int

E → T

E → T + E

E

|int * int + int	**Shift**	T → int
int | * int + int		T → int * T
		T → int
		E → T
		E → T + E
		E

Manchester Metropolitan University

| int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int		T → int * T
		T → int
		E → T
		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int		T → int
		E → T
		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int		T → int
		E → T
		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int	Reduce	T → int
T |+ int		
		E → T
		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int	Reduce	T → int
T |+ int	Shift	
T + |int		E → T
		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int	Reduce	T → int
T |+ int	Shift	
T + |int	Shift	E → T
T + int |		
		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int	Reduce	T → int
T |+ int	Shift	
T + |int	Shift	E → T
T + int |	Reduce	
T + T |		E → T + E
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int	Reduce	T → int
T |+ int	Shift	
T + |int	Shift	E → T
T + int |	Reduce	
T + T |	Reduce	E → T + E
T + E |		
		E

Manchester Metropolitan University

|int * int + int	**Shift**	T → int
int | * int + int	Shift	
int * | int + int	Shift	T → int * T
int * int |+ int	Reduce	
int * T |+ int	Reduce	T → int
T |+ int	Shift	
T + |int	Shift	E → T
T + int |	Reduce	
T + T |	Reduce	E → T + E
T + E |	Reduce	
E |		E

Practical Issues (I)

- Examples presented have not included error recovery – so errors are detected and fixed one-at-a-time : (
- Most simple error recovery is to scan to next semi-colon (assuming a suitable language), and pick up from there.
  - Easy with a top-down parser
  - More challenging (but still possible) with bottom-up

Practical Issues (II)

- Left- versus right-recursion: while top-down parsers don't like left-recursion, bottom-up can handle either form – so which is best?
- Factors to consider:
  - Stack depth (generally less for left-recursive)
  - Associativity (left-recursion produces left-associativity, and right-recursion right-associativity)

In Summary

- Top-down parsing means…
- Bottom-up parsing means…
- Backtracking versus lookahead
- Construction of First and Follow sets
- Basic principles of bottom-up parsing
- Practical issues for consideration

Where we are…

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