Lancaster University

Unit 3:
An Introduction to Parsing Strategies

SCC 312 Compilation
John Vidler
j.vidler@lancaster.ac.uk

Original slide deck by John Mariani

Lancaster University

Aims

- What are we trying to do in syntactic analysis, anyway?
- The search problem : top-down and bottom-up
- Algorithms and efficiency
- Criteria for a parsing strategy
- Finally, LR(k) and LL(k)

Lancaster University

Syntactic Analysis

- The task of the syntactic analysis phase is essentially to build a parse tree
  - With the root : the distinguished symbol, “<program>” or whatever
  - The leaves : the sequence of tokens from the lexical analysis phase
  - ...and with each branch point in the tree ***sanctioned*** by a rule from the grammar

Lancaster University

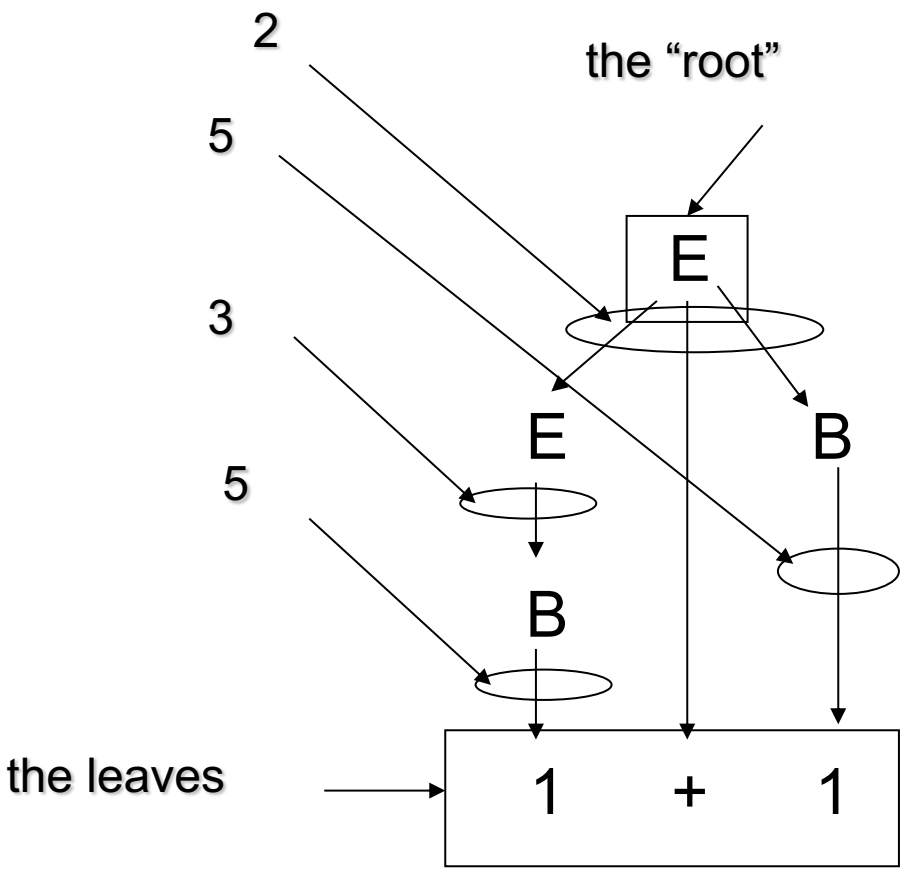
Syntactic Analysis

- Perhaps carry out semantic checks at the same time
- This phase may instead generate a sequence of machine code instructions, or some other intermediate form of the program, a bit at a time
- If the syntax of the program is invalid, to give as many and as helpful error messages as possible

Parse Tree showing rule sanctions

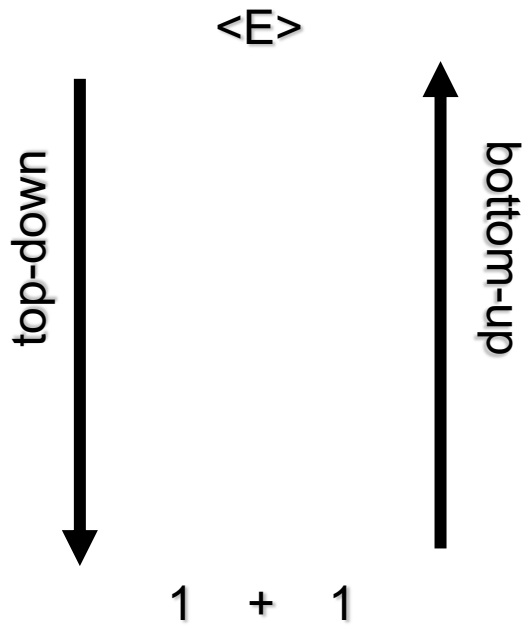
```
E → E + B  (2)
    E + 1  (5)
    B + 1  (3)
    1 + 1  (5)
```

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```



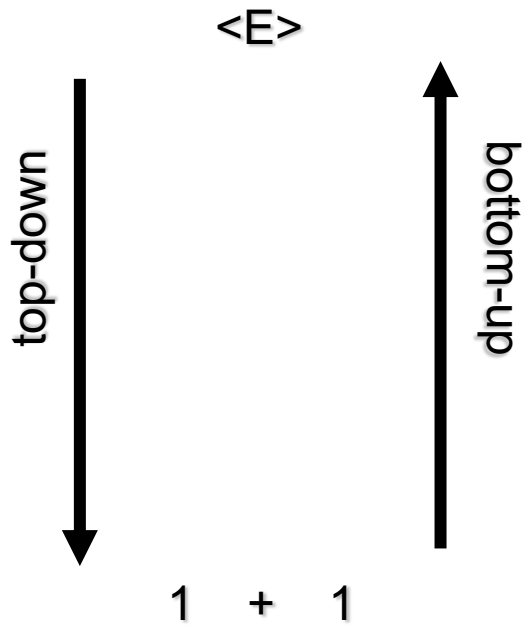
The Search Problem

- So we have a search problem - given the root and leaves, to find the appropriate labeled branch points in between
- We could start the search from the top, from <program>, and try to work down towards the leaves filling in the tree; this is a ***top-down*** parsing strategy



The Search Problem

- Or we could start with the sequence of leaves from the lexical analysis phase, and try to build parts of the tree as we work up towards <program>; this is a ***bottom-up*** parsing strategy



Bottom-Up

```
      1 + 1  (5)
      B + 1  (3)
      E + 1  (5)
E → E + B  (2)
```

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

1	+	1

Bottom-Up

```
      1 + 1  (5)
      B + 1  (3)
      E + 1  (5)
E → E + B  (2)
```

We convert 1 to B, using rule 5 to sanction the transformation.

1	+	1
B	+	1

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Bottom-Up

```
      1 + 1  (5)
      B + 1  (3)
      E + 1  (5)
E → E + B  (2)
```

We convert B to E, using rule 3 to sanction the transformation.

1	+	1
B	+	1
E	+	1

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Bottom-Up

```
      1 + 1  (5)
      B + 1  (3)
      E + 1  (5)
E → E + B  (2)
```

We convert 1 to B, using rule 5 to sanction the transformation.

1	+	1
B	+	1
E	+	1
E	+	B

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Bottom-Up

```
          1 + 1  (5)
          B + 1  (3)
          E + 1  (5)
E → E + B  (2)
```

(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

We convert E + B to E, using rule 2 to sanction the transformation. We have arrived at the root of the grammar, and have therefore finished the parse.

1	+	1
B	+	1
E	+	1
E	+	B
	E	

Top-Down

```
E → E + B  (2)
    E + 1  (5)
    B + 1  (3)
    1 + 1  (5)
```

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

	E	

Top-Down

```
E → E + B  (2)
    E + 1  (5)
    B + 1  (3)
    1 + 1  (5)
```

We convert E to E + B, using rule 2 to sanction the transformation.

	E	
E	+	B

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Top-Down

```
E → E + B  (2)
    E + 1  (5)
    B + 1  (3)
    1 + 1  (5)
```

We convert B to 1, using rule 5 to sanction the transformation.

	E	
E	+	B
E	+	1

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Top-Down

```
E → E + B  (2)
    E + 1  (5)
    B + 1  (3)
    1 + 1  (5)
```

We convert E to B, using rule 3 to sanction the transformation.

	E	
E	+	B
E	+	1
B	+	1

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Top-Down

```
E → E + B  (2)
    E + 1  (5)
    B + 1  (3)
    1 + 1  (5)
```

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

We convert B to 1, using rule 5 to sanction the transformation. We have arrived at the sentence we were checking, and have therefore finished the parse.

	E	
E	+	B
E	+	1
B	+	1
1	+	1

Lancaster University

The Size of the Problem

- It can be shown that, for certain context-free grammars, it can take an exponential amount of time to parse a string
  - input size is the length of the string
- For example, consider the grammar:
  S -> S + S | S * S | a
- Here the number of possible parses grows exponentially with the length of the input string
- So a simple-minded attempt at parsing would be an *exponential* process

Lancaster University

Some Algorithms

- However there are parsing strategies which are guaranteed to do better than this
- *Earley's algorithm* is a top-down strategy for parsing with a context-free grammar, which parses in time n^3 where n is the length of the input string
- Similarly the CYK (*Cocke-Younger-Kasami*) algorithm is a bottom-up strategy which also parses in time n^3, but it requires the context-free grammar to be converted to a special form, *Chomsky Normal Form*

Lancaster University

Earley & CYK

- These two algorithms are guaranteed to do better than the simple-minded (and possibly exponential) approach
- But they both require us to build parts of the tree without knowing exactly how the parts will go to make up the whole
- It would be nice to have parsing strategies which are more efficient still, if that were possible

Criteria for a Parsing Strategy

- Power
- Back-tracking
- Lookahead
- Efficiency
- Small

Lancaster University

Power

- Our strategy must be ***powerful*** enough for the language being parsed
- Earley and CYK can be used on ***any*** context-free grammar
- The parsing strategies we are going to be developing in this course can be used ***only on subsets of the context-free languages***

Lancaster University

Power

- Luckily the subsets can be made to cover most features we are likely to want in a programming language
- But when we write a grammar for a new programming language, we must check that the grammar is compatible with the proposed parsing strategy
- If not, we have to revise the grammar in some suitable way

Lancaster University

Back-Tracking

- Some parsing strategies involve back-tracking
- When the parser decides that an earlier decision about the form of some part of the parse tree was wrong, it goes back to that decision and retakes it in a different way
- Back-tracking is inefficient, as it requires the parser to make a number of passes over the input stream

Lancaster University

Back-Tracking

- But also any consequences of the decision have to be reversed in some way
  - entries removed from the symbol table, or modified
  - generated machine/intermediate code deleted
  - etc.
- The techniques in this course require ***no*** back-tracking
- Once a decision has been made about the form of some part of the input program, this will never be revised

Lancaster University

Lookahead

- An alternative to back-tracking is ***lookahead***; when the parser needs to make a decision, it looks at the next few tokens in the input stream in order to help make the correct decision
- In principle the parser could look arbitrarily far along the input program, perhaps right to the end, but this is likely to be inefficient
  - So we want to minimise the amount of lookahead

Lancaster University

Lookahead

- Both the strategies in this course involve some amount of lookahead, but they are usually implemented to require a maximum of ***one symbol of lookahead***
  - That is, the parser is allowed to look only at the *next* input token from the lexical analyser when making a decision. This is generally sufficient for normal programming languages

Lancaster University

Efficiency

- We want our parser to be *efficient*; that is, to require time proportional to the length of the program
  - we are unlikely to be able to do better than that
- The parsing strategies here are of this type
  - that is, they are O(n) algorithms, where n is the length of the input string

Lancaster University

Small Size

- It would be nice if the parser could also be *small*, but this is less of a problem nowadays, as storage is so cheap
- Some versions of the best parsing strategy can in fact generate very large tables to specify to the parser what to do in each situation. This would have to be avoided if we are working in a small machine

Lancaster University

LR (k) and LL (k)

The two parsing strategies we will examine

LL(k) and LR(k) Parsers

- We will be looking at two main sets of parsing strategies, called LL(k) and LR(k)
- The first L means that the input stream of tokens is processed from left to right
- The k is the number of symbols (tokens) of lookahead we are allowed; it will turn out that this is usually 0 or 1

$$L_L(k)$$

$$L_R(k)$$

LL(k) and LR(k) Parsers

- The second letter specifies the type of derivation produced by the parser
- An LL(k) parser generates a left-most derivation, and is a top-down parser
- An LR(k) parser generates a right-most derivation, and is a bottom-up method

L L (k)

L R (k)

Learning Outcomes

- You should now understand
  - That the parsing process is a search problem
  - Efficiency is, as always, an issue
  - The desirable properties for a parsing strategy
  - LL(k) and LR(k) are the two main strategies we will explore – not as complete as Earley and CYK but fortunately complete enough to deal with most PL grammars

THE END