Compilers

Topic 3: Syntactic analysis (LR)

Mick O'Donnell: michael.odonnell@uam.es

Topic 3: Syntactic analysis (LR)

3.1 Introduction to parsing
3.1 Introduction to Parsing

Main function of parser:
- Produce a parse tree which is then used by Code Generator to produce target code

Secondary function of parser:
- Syntactic error detection – report to user where any error in the source code are.

The parser needs to be designed to match both these functions

The design of the parser could be simpler if only compilation was needed:
- If debugging not an issue, parser could stop at first instance of malformed input
- However, to optimise the Code/Compile/Debug cycle, the compiler should not stop on the first detected syntax error, but rather, produce a listing of all errors

Topics in Parsing

1. Notion of grammar
   - Terminals
   - Nonterminals
   - Start Symbol

2. Grammar Rules
   - A -> B C
   - A -> B | C

3. Applying a grammar (building a parse tree)
   - Grammar: E -> E + E | E * E | -E | (E) | id
   - Example: a + b
   - Example: a * (b + c)
   - good parse tree has start symbol at top
3.1 Introduction to parsing

3. Applying a grammar (cont.)

- Grammar:
  - $E \rightarrow E + E$
  - $E \rightarrow E \times E$
  - $E \rightarrow -E$
  - $E \rightarrow (E)$
  - $E \rightarrow \text{id}$

- Example 1: $a + b$
3.1 Introduction to parsing

- Grammar:
  - E -> E + E
  - E -> E * E
  - E -> -E
  - E -> (E)
  - E -> id

- Example 1: a + b

```
E
   |
 a + b
   ↑
```
3.1 Introduction to parsing

- Grammar:
  - $E \rightarrow E + E$
  - $E \rightarrow E \times E$
  - $E \rightarrow -E$
  - $E \rightarrow (E)$
  - $E \rightarrow \text{id}$

- Example 1: $a + b$

```
E     E
|     |
a   +   b
```

3.1 Introduction to parsing

- Grammar:
  - $E \rightarrow E + E$
  - $E \rightarrow E \times E$
  - $E \rightarrow -E$
  - $E \rightarrow (E)$
  - $E \rightarrow \text{id}$

- Example 1: $a + b$

```
\downarrow
E     E
|     |
a   +   b
```
3.1 Introduction to parsing

- Grammar:
  - $E \rightarrow E + E$
  - $E \rightarrow E \ast E$
  - $E \rightarrow -E$
  - $E \rightarrow (E)$
  - $E \rightarrow id$

- Example 1: $a + b$

```
    E
   / | \
  E  E
 / |   |
a + b
```

- All input tokens incorporated
- Top token is start token
- Thus, a good parse

4. Order of application is important
- Parse example: $a + b \ast c$
- Left-right application
- Right to left application
3.1 Introduction to parsing

4. Order of application is important
   • Parse example: a + b * c
   • Left-right application
   • Right to left application

\[
E \\
| a + b * c
\]

Grammar:
\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E \\
E & \rightarrow -E \\
E & \rightarrow (E) \\
E & \rightarrow id
\end{align*}
\]
4. Order of application is important
   • Parse example: \( a + b \times c \)
   • **Left-right application**
   • Right to left application

**Grammar:**

\[
\begin{align*}
  E & \rightarrow E + E \\
  E & \rightarrow E \times E \\
  E & \rightarrow -E \\
  E & \rightarrow (E) \\
  E & \rightarrow id
\end{align*}
\]
3.1 Introduction to parsing

4. Order of application is important
   • Parse example: $a + b \times c$
   • **Left-right application**
   • Right to left application

Grammar:

- $E \rightarrow E + E$
- $E \rightarrow E \times E$
- $E \rightarrow -E$
- $E \rightarrow (E)$
- $E \rightarrow id$

---

4. Order of application is important
   • Parse example: $a + b \times c$
   • **Left-right application**
   • Right to left application

Grammar:

- $E \rightarrow E + E$
- $E \rightarrow E \times E$
- $E \rightarrow -E$
- $E \rightarrow (E)$
- $E \rightarrow id$
3.1 Introduction to parsing

4. Order of application is important
   • Parse example: \(a + b \times c\)
   • Left-right application
   • Right to left application

```
Grammar:
E -> E + E
E -> E * E
E -> -E
E -> (E)
E -> id
```

```
E
  /|
E  E
  /|
a  +  b  *  c
```

```
E
  /|
E  E
  /|
a  +  b  *  c
```

```
Grammar:
E -> E + E
E -> E * E
E -> -E
E -> (E)
E -> id
```

```
E
  /|
E  E
  /|
a  +  b  *  c
```

```
E
  /|
E  E
  /|
a  +  b  *  c
```
4. Order of application is important
- Parse example: $a + b * c$
- **Left-right application**
- Right to left application

**Grammar:**
- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow -E$
- $E \rightarrow (E)$
- $E \rightarrow id$

E \[\begin{array}{c}
E \\
\mid \\
a + \\
\end{array}\]

E \[\begin{array}{c}
E \\
\mid \\
E \\
\mid \\
b * \\
\mid \\
c \\
\end{array}\]

E \[\begin{array}{c}
E \\
\mid \\
a + \\
\end{array}\]

E \[\begin{array}{c}
E \\
\mid \\
E \\
\mid \\
b * \\
\mid \\
c \\
\end{array}\]

E \[\begin{array}{c}
E \\
\mid \\
a + \\
\end{array}\]

E \[\begin{array}{c}
E \\
\mid \\
E \\
\mid \\
b * \\
\mid \\
c \\
\end{array}\]
3.1 Introduction to parsing

4. Order of application is important
   • L-R and R-L application give different trees
   • Semantics may differ

5. Rewrite rules and Derivations (IMPORTANT)
   • Concept of derivation: being sequence of rewrites from start symbol to surface structure
   • Parse tree is a graphical representation of a derivation sequence

6. Handling Ambiguous Parses (e.g., $a + b * c$)
   • Two approaches:
     • Add disambiguating rules that throw away undesirable parse trees, leaving just one
     • Rewrite grammar to be unambiguous
6. Handling Ambiguous Parses (cont.)

- The dangling else problem:

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
\text{stmt} \rightarrow \text{if expr then stmt else stmt}
\]

But consider the code:

```java
if x==1 then if y==2 print 1 else print 2
```

OR

```java
if x==1 then [if y==2 print 1 else print 2]
```

6. Handling Ambiguous Parses (cont.)

- Rewriting the grammar to remove ambiguity:

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
\text{stmt} \rightarrow \text{if expr then stmt else stmt}
\]

\[
\text{stmt} \rightarrow \text{matched_stmt} | \text{unmatched_stmt} | \text{other_stmt} \\
\text{matched_stmt} \rightarrow \text{if expr then matched_stmt else matched_stmt} | \text{other_stmt} \\
\text{unmatched_stmt} \rightarrow \text{if expr then stmt} | \text{if expr then matched_stmt else unmatched_stmt}
\]
3.1 Introduction to parsing

7. Top-down vs. Bottom-up analysis

- Previously, we have been building trees by bottom-up application of rules
  - bottom-up parsing is a parsing method that works by identifying terminal symbols first, and combines them successively to produce nonterminals
  - We build structure from the bottom up.

- Other approaches build structure from the top-down:
  - We start with the START symbol
  - We apply expansions of non-terminal symbols
7. Top-down vs. Bottom-up analysis

Apply: $E \rightarrow E + E$

Apply: $E \rightarrow \text{id}$
7. Top-down vs. Bottom-up analysis

Apply: $E \rightarrow E \cdot E$

Apply: $E \rightarrow id$
7. Top-down vs. Bottom-up analysis

```
Apply: E -> id
```

3.1 Introduction to parsing

10. Types of parsers

- Top Down:
  - LL parsers
- Bottom Up
  - LR parsers
    - LR(0)
    - SLR(1) (Simple LR)
    - LR(1) (Canonical LR)
    - LALR (LookAhead LR)
    - LR(k)
- Operator Precedence Parsers
Derivation Sequences

Derivations

• A ‘derivation’ displays the sequence of substitutions from the START symbol to the input.

• A ‘leftmost derivation’ is one which, working from top to bottom, the leftmost nonterminal is the one to expand

```
E
E * E
(E) * E
(E + E) * E
(id + E) * E
(id + id) * E
(id + id) * id
```

Derivation Sequences

Rightmost Derivations

• A ‘rightmost derivation’ is one which, working from top to bottom, the rightmost nonterminal is the one expanded

```
E
E * E
E * id
(E) * id
(E + E) * id
(E + id) * id
(id + id) * id
```
LR Parsers and Derivations

- The main family of Bottom-Up parsers
- An LR parser is one which:
  - **Left to right processing of input**
  - **Rightmost derivation**
- Now, this last *seems* to mean it attempts to apply rules on the **right** side of the input.
- But in fact, this is not so: rules are applied to the **left side** of the structure first.
- But when we look at the derivation tree produced, it represents a rightmost derivation when viewed top-down.

**Derivation Sequences**

- NOTE: Leftmost application of rules in BU processing produces a rightmost derivation

```
E
E * E
E * id
( E ) * id
( E + E ) * id
( E + id ) * id
( id + id ) * id
```
Topic 3: Syntactic analysis

3.2 Shift-Reduce Parsers

Mick O'Donnell: michael.odonnell@uam.es

Shift-Reduce Parsing
• A bottom-up parsing technique
• Used in LR parsers
• Two basic concepts:
  • Shift
  • Reduce
Reducing

- The main concept here is that we have a stack (‘pila’) which holds the tokens we have so far recognised.
- Where the tokens on the top of the stack match the RHS of a rule, we can replace those tokens with the LHS of the rule (this operation is called ‘reduce’).

```
Stack: ( E + E)  reduce  Stack: ( E)

E -> E + E
```

Shifting

- When we cannot reduce the stack, we shift in another input token:

```
Stack: ( E)  shift  Stack: ( E)

Input: ( id + id) * id
```
LR analysis

Shift-Reduce Parsing

Shift-Reduce Cycle
- We start with an empty stack
- We shift in the first token
- We then start a cycle:
  1. If the top of stack matches a RHS
     - Reduce
  2. Else:
     - Shift
  3. Goto (1)
- We terminate when the input is exhausted.

Input: 

Stack:

Pointer at next input token:

List of input tokens:

Input: ( id + id ) * id

Stack of matched terminals or reductions of terminals
Input: \(( id + id ) * id\)

Stack: 

Action: Shift

**Shift:** move next token to the stack and advance pointer
Input: \(( \text{id} + \text{id} ) \ast \text{id}\)

Stack: ()

Query: Can reduce?

Grammar:
- \(E \rightarrow E + E\)
- \(E \rightarrow E \ast E\)
- \(E \rightarrow \neg E\)
- \(E \rightarrow (E)\)
- \(E \rightarrow \text{id}\)

Response: No.
Input: \((\text{id} + \text{id}) \ast \text{id}\)

Stack: \[
\]

Grammar:

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E \\
E & \rightarrow -E \\
E & \rightarrow (E) \\
E & \rightarrow \text{id}
\end{align*}
\]

Query: Can reduce?
Response: No.
Action: Shift
Input: \(( \text{id} + \text{id} ) \times \text{id}\)

Stack: \(\text{id}\)

Grammar:
- \(E \rightarrow E + E\)
- \(E \rightarrow E \times E\)
- \(E \rightarrow -E\)
- \(E \rightarrow (E)\)
- \(E \rightarrow \text{id}\)

Query: Can reduce?
Response: Yes.
Action: Reduce

We can reduce when the top elements in the stack match the RHS of a rule.

No change of input pointer

Input: \(( \text{id} + \text{id} ) \times \text{id}\)

Stack: \(\text{E}\)

Grammar:
- \(E \rightarrow E + E\)
- \(E \rightarrow E \times E\)
- \(E \rightarrow -E\)
- \(E \rightarrow (E)\)
- \(E \rightarrow \text{id}\)

\(\text{id}\) replaced by \(\text{E}\)
Input: \( (\text{id} + \text{id}) \ast \text{id} \)

Stack: \( \text{[E]} \)

Query: Can reduce?

Response: No.

Action: Shift
Input: \((id + id) \times id\)

Stack: \(\text{E} +\)

Query: Can reduce?
Response: No.
Action: Shift

Input: \((id + id) \times id\)

Stack: \(\text{E} + id\)

Query: Can reduce?
Response: Yes.
Action: Reduce
Input: \((\text{id} + \text{id}) \times \text{id}\)

Stack: \(\text{( E } + \text{ E)}\)

Grammar:
\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \times E \\
E & \rightarrow -E \\
E & \rightarrow (E) \\
E & \rightarrow \text{id}
\end{align*}
\]

Query: Can reduce?
Response: Yes.
Action: Reduce
Input: \((id + id) \times id\)

Stack: \( (E) \)

Query: Can reduce?
Response: Yes.
Action: Reduce

Grammar:
- \(E \rightarrow E + E\)
- \(E \rightarrow E \times E\)
- \(E \rightarrow -E\)
- \(E \rightarrow (E)\)
- \(E \rightarrow id\)
Input: $\left( \text{id} + \text{id} \right) \times \text{id}$

Stack: $E$

Grammar:

- $E \rightarrow E + E$
- $E \rightarrow E \times E$
- $E \rightarrow -E$
- $E \rightarrow (E)$
- $E \rightarrow \text{id}$
Input: $(id + id) \times id$

Stack: $E \times id$

Grammar:
- $E \rightarrow E + E$
- $E \rightarrow E \times E$
- $E \rightarrow -E$
- $E \rightarrow (E)$
- $E \rightarrow id$

Input: $(id + id) \times id$

Stack: $E \times E$

Grammar:
- $E \rightarrow E + E$
- $E \rightarrow E \times E$
- $E \rightarrow -E$
- $E \rightarrow (E)$
- $E \rightarrow id$
Input: \((\text{id} + \text{id}) \times \text{id}\)

Stack: \[E\]

Grammar:
\[
\begin{align*}
E &\rightarrow E + E \\
E &\rightarrow E \times E \\
E &\rightarrow -E \\
E &\rightarrow (E) \\
E &\rightarrow \text{id}
\end{align*}
\]

At this point, input is exhausted, and stack contains START symbol. A successful parse.

Query: Input exhausted?
Response: Yes.
Query: Stack == Start Symb?
Action: Accept
The four actions of a Shift-Reduce parser are:

- **Shift** – move next input onto stack and advance input pointer
- **Reduce** – replace symbols on top of stack with rule LHS
- **Accept** - the parser announces successful completion of parsing;
- **Error** - the parser discovers that a syntax error has occurred.