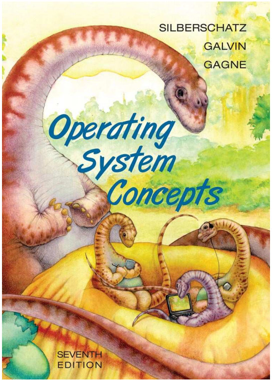
Syntax Analysis - Parsing

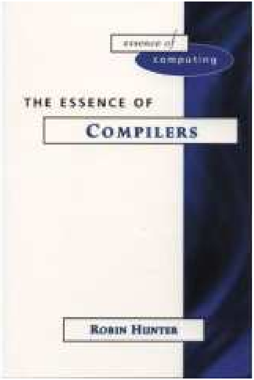
Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems

Uppsala University

Spring 2008





Operating Systems

- Process Management
- Memory Management
- Storage Management

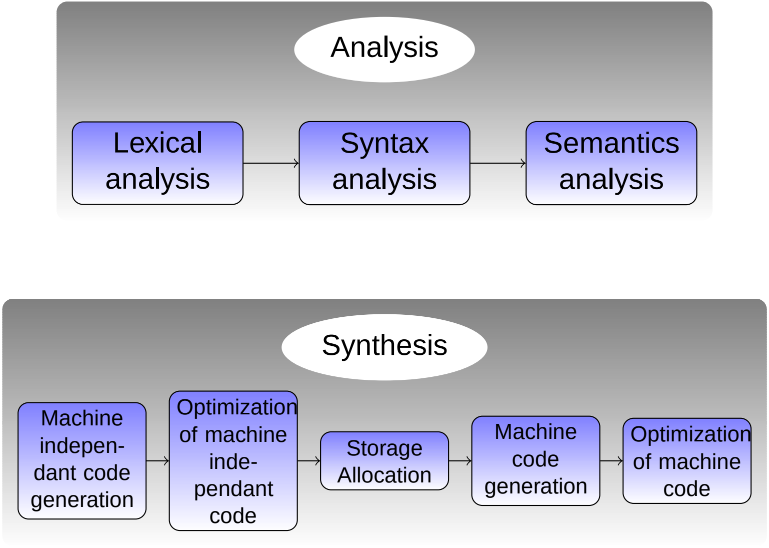
Compilers

- Compiling process & Lexical analysis
- Parsing
- Semantic & Code generation

Recall – Lexical Analysis

- Transforms the stream of characters into tokens
- Uses regular expression to validate tokens
- Uses Finite Automata for transformation mechanism

- Lexical Analysers refered as lexers


Analysis
Lexical analysis
Syntax analysis
Semantics analysis
Synthesis
Machine indepen-dant code generation
Optimization of machine inde-pendant code
Storage Allocation
Machine code generation
Optimization of machine code

Syntax Analysis

Goal

Identify if (input) token streams satisfy the program syntax

We need:

- Expressive way to describe the syntax
- Acceptor that determine if the token streams satisfy the syntax of the program

For Lexical analysis:

- Regular expressions, to describe tokens
- Finite Automata, as acceptors for regular expressions

Regular Expressions?

Why not using RE again but, this time, on tokens?

Reason: Not enough power to express the syntax in programming languages

Example: Nested constructs like Blocks, Expressions, Statements.

Solution: Use Context-Free Grammars

Context-Free Grammars

- Terminal symbols: token or ϵ
- Non-terminal symbols: syntactic variables
- Start Symbol S: special non-terminal
- Productions of the form $LHS \rightarrow RHS$
  - LHS: a single non-terminal
  - RHS: a string of terminals and non-terminals
  - Specifies how non-terminals may be expanded

Example

- $S \rightarrow a\ S\ a$
- $S \rightarrow T$
- $T \rightarrow b\ T\ b$
- $T \rightarrow \epsilon$

Example: Balanced-parenthesis

Grammar for balanced parenthesis:

1. $S \to \{\ S\ \}\ S$
2. $S \to \epsilon$

If a grammar accepts a string, there is a derivation of that string using productions.

Example (String { { } })

$S \to \{\ S\ \}\ S \to \{\ S\ \}\ \epsilon \to \{\ \{\ S\ \}\ S\ \}\ \epsilon \to \{\ \{\ S\ \}\ \epsilon\ \}\ \epsilon \to \{\ \{\ \epsilon\ \}\ \epsilon\ \}\ \epsilon$

Short-Hand notation

- $S \to a\ S\ a$
- $S \to T$
- $T \to b\ T\ b$
- $T \to \epsilon$

$\Rightarrow$

- $S \to a\ S\ a \mid T$
- $T \to b\ T\ b \mid \epsilon$

Derivation order

2 standard orders: left-most and right-most

Left-most derivation: in the string, find the left-most non-terminal and apply a production

$$E + S \rightarrow 1 + S$$

Right-most derivation: in the string, find the right-most non-terminal and apply a production

$$E + S \rightarrow E + E + S$$

Grammar for Sum

- $S \rightarrow E + S \mid E$
- $E \rightarrow number \mid (\, S \,)$

Expanded:

1. $S \rightarrow E + S$
2. $S \rightarrow E$
3. $E \rightarrow number$
4. $E \rightarrow (\, S \,)$

Example of accepted input:
$(1 + 2 + (3 + 4)) + 5$

Derivation Example

Example (Derivation of $(1 + 2 + (3 + 4)) + 5$)

$$
\begin{aligned}
S &\rightarrow E + S \rightarrow (S) + S \rightarrow (E + S) + S \rightarrow \\
&\rightarrow (1 + S) + S \rightarrow (1 + E + S) + S \rightarrow \\
&\rightarrow (1 + 2 + S) + S \rightarrow (1 + 2 + E) + S \rightarrow \\
&\rightarrow (1 + 2 + (S)) + S \rightarrow (1 + 2 + (E + S)) + S \rightarrow \\
&\rightarrow (1 + 2 + (3 + S)) + S \\
&\rightarrow (1 + 2 + (3 + E)) + S \\
&\rightarrow (1 + 2 + (3 + 4)) + S \\
&\rightarrow (1 + 2 + (3 + 4)) + E \\
&\rightarrow (1 + 2 + (3 + 4)) + 5
\end{aligned}
$$

Parse Tree vs Abstract Syntax Tree

Concrete Syntax Tree = Parse Tree

Ambigous grammar

Example

$S \to S + S \mid S * S \mid number$

Different derivation produce different parse trees

Expression: 1 + 2 * 3

Derivation 1:

$S \to S + S \to 1 + S \to 1 + S * S \to 1 + 2 * S \to 1 + 2 * 3$

Derivation 2:

$S \to S * S \to S * 3 \to S + S * 3 \to S + 2 * 3 \to 1 + 2 * 3$

Eliminating ambiguity

- By adding non-terminals
- By allowing recursion to the right only, *or* to the left only

$$S \rightarrow S + S \mid S * S \mid number \quad \Rightarrow \quad \begin{array}{l} S \rightarrow S + T \mid T \\ T \rightarrow T * number \mid number \end{array}$$

Conclusion on Grammars

- Context-Free Grammar allow concise syntax specification of programming languages
- A CFG specifies how to convert token stream to parse tree (if non ambiguous!)

Parsing Top-Down

Goal

Construct a derivation of a string,
while reading in the token stream

Top-Down = Left-most

We start from the start symbol and
generate the sentence

Bottom-Up = Right-most

We start from the sentence and
reduce it to the start symbol

Top-Down Lookahead

Want to decide which production to apply based on the next symbols

- $\{x^m y^n \mid m, n > 0\}$
- $S \rightarrow XY$
- $X \rightarrow xX$
- $X \rightarrow x$
- $Y \rightarrow yY$
- $Y \rightarrow y$

Generate xxxyyy:

$S \rightarrow XY \rightarrow xXY \rightarrow xxXY \rightarrow xxxY \rightarrow xxxyY \rightarrow xxxyy$

Top-Down Lookahead

At most stages of the derivation, knowledge was required of *two* symbols beyond those generated so far

Wish

Seek grammars which require at most a single symbol of lookahead at each stage of the derivation, in order to identify the correct production to apply

LL(1)

Left-to-right scanning, Left-most derivation, 1 lookahead symbol

Bad example

Example (Bad)

- $S \rightarrow E + S \mid E$
- $E \rightarrow \textit{number} \mid (\; S \;)$

(1)

$S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$

(1)+2

$S \rightarrow E + S \rightarrow (S) + S \rightarrow$
$(E) + S \rightarrow (1) + S \rightarrow (1) + 2$

Making the grammar LL(1)

Problem: Can't decide which S production to apply until we see symbol after the expression

Left-factoring: Factor common S prefix and add a new non-terminal S' at the decision point

$$S \rightarrow E + S$$
$$S \rightarrow E$$
$$E \rightarrow number$$
$$E \rightarrow (S)$$

$$S \rightarrow E S'$$
$$S' \rightarrow +S$$
$$S' \rightarrow \epsilon$$
$$E \rightarrow number$$
$$E \rightarrow (S)$$

Predictive Parsing

For LL(1) grammar

For a given non-terminal, the lookahead symbol determines uniquely the production to apply

Top-Down parsing = Predictive parsing

S	(	(1+2+(3+4))+5
→ ES'	(	(1+2+(3+4))+5
→ (S)S'	1	(1+2+(3+4))+5
→ (ES')S'	1	(1+2+(3+4))+5
→ (1S')S'	+	(1+2+(3+4))+5
→ (1+S)S'	2	(1+2+(3+4))+5
→ (1+ES')S'	2	(1+2+(3+4))+5
→ (1+2S')S'	+	(1+2+(3+4))+5

Using table

S	(	(1+2+(3+4))+5
→ ES’	(	(1+2+(3+4))+5
→ (S)S’	1	(1+2+(3+4))+5
→ (ES’)S’	1	(1+2+(3+4))+5
→ (1S’)S’	+	(1+2+(3+4))+5
→ (1+S)S’	2	(1+2+(3+4))+5
→ (1+ES’)S’	2	(1+2+(3+4))+5
→ (1+2S’)S’	+	(1+2+(3+4))+5

	number	+	(	)	⊥
S	$\rightarrow ES'$		$\rightarrow ES'$		
S’		$\rightarrow +S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
E	$\rightarrow number$		$\rightarrow (S)$		

Recursive-Descent Parser

	number	+	(	)	$\perp$
$\hookrightarrow S$	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
E	$\rightarrow$ *number*		$\rightarrow (S)$		

```
void parse_S(){
  switch(token){
    case number:parse_E();parse_S'();return;
    case '(':parse_E();parse_S'();return;
    default: error();
  }
}
```

Recursive-Descent Parser

	number	+	(	)	$\perp$
S	$\to ES'$		$\to ES'$		
$\hookrightarrow$S'		$\to +S$		$\to \epsilon$	$\to \epsilon$
E	$\to$ *number*		$\to (S)$		

```
void parse_S'(){
  switch(token){
    case '+':token=input.read();parse_S();return;
    case ')':return;
    case EOF: return;
    default: error();
  }
}
```

Recursive-Descent Parser

	number	+	(	)	$\perp$
S	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
$\hookrightarrow$E	$\rightarrow$ *number*		$\rightarrow (S)$		

```
void parse_E(){
  switch(token){
    case number:token = input.read();return;
    case '(':token=input.read();parse_S();
             if(token != ')')error();
     token = input.read();return;
    default: error();
  }
}
```

Parse table

$$\underbrace{\textit{Grammar} \Rightarrow \textit{Parse Table}}$$

For every non-terminal, every lookahead symbol can be handled by at most one production

Grammar is LL(1) = no conflicting entries in the table

Example (Ambiguous ⇒ Conflicts)

$S \rightarrow S + S \mid S * S \mid number$

	number	+	*
S	$\rightarrow number, \rightarrow S + S, \rightarrow S * S$		

Summary

LL(k) grammar

- left-to-right scanning
- left-most derivation
- can determine what production to apply from the next *k* symbols
- Can automatically build predictive parsing tables

Predictive Parsers

- Can be easily built for LL(k) grammars from parsing tables
- Also called recursive-descent or top-down parsers

So far

- Have been using grammar for language of "sums with parenthesis": (1+2+(3+4))+5
- Started with simple, right-associative grammar:
  - $S \rightarrow E+S \mid E$
  - $E \rightarrow number \mid (S)$
- Transformed it into LL(1) grammar by left-factoring:
  - $S \rightarrow ES'$
  - $S' \rightarrow \epsilon \mid +S$
  - $E \rightarrow number \mid (S)$
- What if we start with left-associative grammar?
  - $S \rightarrow E+S \mid E$
  - $E \rightarrow number \mid (S)$

Left vs Right associative

Right-recursion: right-associative

- $S \rightarrow E + S \mid E$
- $E \rightarrow number$

Left-recursion: left-associative

- $S \rightarrow E + S \mid E$
- $E \rightarrow number$

Problem

Left-recursive grammar are not LL(1)

There exists an algorithm for left-recursion elimination:
Left-recursion $\Rightarrow$ Right-recursion

Creating LL(1) grammar

1. Start with left-recursive grammar:
   - $S \rightarrow S+E$
   - $S \rightarrow E$
2. Apply left-recursion elimination:
   - $S \rightarrow ES'$
   - $S' \rightarrow +ES' \mid \epsilon$
3. Start with right-associative grammar
   - $S \rightarrow E+S$
   - $S \rightarrow E$
4. Apply left-factoring to eliminate common prefixes:
   - $S \rightarrow ES'$
   - $S' \rightarrow +S \mid \epsilon$

Top-Down Parsing Summary

Language Grammar

$\Downarrow$

Left-recursion elimination / left-factoring

$\Downarrow$

LL(1) Grammar

$\Downarrow$

Predictive parsing table

$\Downarrow$

Recursive-descent parser

$\Downarrow$

Parser with AST generation

Bottom-Up Parsing

- More powerfull
- LR grammars – more expressive than LL
  - construct right-most derivations
  - left-recursive: virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - Automatic parser generators, like YACC

Parsing Top-Down

Goal

Construct a derivation of a string,
while reading in the token stream

Top-Down = Left-most

We start from the start symbol and
generate the sentence

Bottom-Up = Right-most

We start from the sentence and
reduce it to the start symbol

Backwards...

Start with tokens and end with the start symbol

- $S \rightarrow S + E \mid E$
- $E \rightarrow number \mid (S)$

(1 + 2 + (3 + 4)) + 5 ← (E + 2 + (3 + 4)) + 5
← (S + 2 + (3 + 4)) + 5 ← (S + E + (3 + 4)) + 5
← (S + (3 + 4)) + 5 ← (S + (E + 4)) + 5
← (S + (S + 4)) + 5 ← (S + (S + E)) + 5
← (S + (S)) + 5
← (S + E) + 5
← (S) + 5
← E + 5
← S + 5
← S + E
← S

⇑ Right-most derivation

Advantages

Advantages of bottom-up parsing

Can postpone the selection of productions until more of the input is scanned

Example

- $\{x^m y^n \mid m,n > 0\}$
- $S \rightarrow XY$
- $X \rightarrow xX$
- $X \rightarrow x$
- $Y \rightarrow yY$
- $Y \rightarrow y$

Generate xxxyyy:

$S \rightarrow XY \rightarrow XyY \rightarrow Xyy \rightarrow xXyy \rightarrow xxXyy \rightarrow xxxyy$

Recall with top-down/left-most:

$S \rightarrow XY \rightarrow xXY \rightarrow xxXY \rightarrow xxxY \rightarrow xxxyY \rightarrow xxxyy$

Example

Generate xxxyyy:

$S \rightarrow XY \rightarrow XyY \rightarrow Xyy \rightarrow xXyy \rightarrow xxXyy \rightarrow xxxyy$

$xxxyy \rightarrow xxXyy \rightarrow xXyy \rightarrow Xyy \rightarrow XyY \rightarrow XY \rightarrow S$

In bottom-up parsing, right sides of productions are not recognized until they have been completely read
$\Rightarrow$ Need to store partially recognized right sides (until replacable): a Stack

Bottom-Up information = Information like in top-down + Stack

Shift-Reduce Parsing

Parsing

Sequence of shift and reduce

- Shift: Move lookahead token to stack

Stack	Input	Action
(	1+2+(3+4))+5	Shift 1
(1	+2+(3+4))+5	

- Reduce: Replace symbol γ from top of stack with non-terminal symbol X, corresponding to production $X \longrightarrow \gamma$ (pop γ, push X)

Stack	Input	Action
(S+E	+(3+4))+5	Reduce $S \rightarrow S + E$
(S	+(3+4))+5	

$S \to S + E \mid E \qquad E \to number \mid (S)$

Derivation	Stack	Input	Action
(1+2+(3+4))+5 $\leftarrow$		(1+2+(3+4))+5	Shift
(1+2+(3+4))+5 $\leftarrow$	(	1+2+(3+4))+5	Shift
(1+2+(3+4))+5 $\leftarrow$	(1	+2+(3+4))+5	Reduce $E \to number$
(E+2+(3+4))+5 $\leftarrow$	(E	+2+(3+4))+5	Reduce $S \to E$
(S+2+(3+4))+5 $\leftarrow$	(S	+2+(3+4))+5	Shift
(S+2+(3+4))+5 $\leftarrow$	(S+	2+(3+4))+5	Shift
(S+2+(3+4))+5 $\leftarrow$	(S+2	+(3+4))+5	Reduce $E \to number$
(S+E+(3+4))+5 $\leftarrow$	(S+E	+(3+4))+5	Reduce $S \to S + E$
(S+(3+4))+5 $\leftarrow$	(S	+(3+4))+5	Shift
(S+(3+4))+5 $\leftarrow$	(S+	(3+4))+5	Shift
(S+(3+4))+5 $\leftarrow$	(S+(	3+4))+5	Shift
(S+(3+4))+5 $\leftarrow$	(S+(3	+4))+5	Reduce $E \to number$

⋮

Problem

How do we know which action to take?
Shift or Reduce?
Which production?

Issues:

- Sometimes can reduce but shouldn't (shift-reduce conflict)
- Sometimes can reduce in different ways (reduce-reduce conflict)

Solution

We have algorithms to determine which actions to take

We can construct parsing tables (like top-down but different shapes) and we check for conflicts.

We have theoretical results like

Any language which is LR(k) for a given k is also LR(1)

No need to consider lookaheads of more than one symbol

We have automated tools to do it: **YACC**.

Works like **Lex** and can be combined

Overall

LR parsing has the following features:

- May be applied to a wide class of grammars and languages
- Grammar transformations are usually minimal
- The analysis time is linear in the length of the input
- Syntax errors discovered on the first inadmissible symbol
- It is well supported by tools

Recall – Goal

Goal

Identify if (input) token streams satisfy the program syntax

We need

- Expressive way to describe the syntax
  $\Rightarrow$ LL(1) and LR(1) grammars, ...
- Acceptor that determine if the token streams satisfy the syntax of the program
  $\Rightarrow$ Recursive-Descent and Shift-Reduce Parsers