Compilation 2009

Context-Free Languages Parsers and Scanners

Johnni Winther
Michael I. Schwartzbach
Aarhus University

Context-Free Grammars

Example:

sentence → *subject verb object*
subject → *person*
person → `John | Joe | Zacharias`
verb → `asked | kicked`
object → *thing* | *person*
thing → `the football | the computer`

- **Nonterminal** symbols:
  sentence, subject, person, verb, object, thing
- **Terminal** symbols:
  John, Joe, Zacharias, asked, kicked, the football, the computer
- **Start** symbol: *sentence*
- Example of **derivation**:
  sentence ⇒ *subject verb object* ⇒ ... ⇒ `John asked the computer`

Formal Definition of CFGs

A *context-free grammar* (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$

- V is a finite set of **nonterminal** symbols
- Σ is an alphabet of **terminal** symbols and $V \cap \Sigma = \emptyset$
- $S \in V$ is a **start** symbol
- P is a finite set of **productions**
  of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$

Derivations

- "$\Rightarrow$" denotes a single derivation step, where a nonterminal is rewritten according to a production
- Thus "$\Rightarrow$" is a relation on the set $(V \cup \Sigma)^*$
- If $\alpha, \beta \in (V \cup \Sigma)^*$ then $\alpha \Rightarrow \beta$ when
  - $\alpha = \alpha_1 A \alpha_2$ where $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ and $A \in V$
  - the grammar contains the production $A \rightarrow \gamma$
  - $\beta = \alpha_1 \gamma \alpha_2$

The Language of a CFG

- Define the relation "$\Rightarrow^*$" as the *reflexive transitive closure* of "$\Rightarrow$", that is:

$$\alpha \Rightarrow^* \beta \text{ iff } \alpha \underbrace{\Rightarrow \ldots \Rightarrow \ldots \Rightarrow}_{\text{0 or more steps}} \beta$$

- The **language** of G is defined as $L(G) = \{ x \in \Sigma^* \mid S \Rightarrow^* x \}$
- A language $L \subseteq \Sigma^*$ is **context-free** iff there is a CFG G where $L(G)=L$

Example 1

The language $L = \{ a^n b^n \mid n \geq 0 \}$ is described by a CFG $G=(V,\Sigma,S,P)$ where

- $V = \{S\}$
- $\Sigma = \{a,b\}$
- $P = \{S \rightarrow aSb,\ S \rightarrow \Lambda\}$

That is, $L(G) = L$

alternative notation:
$S \rightarrow aSb \mid \Lambda$

Example 2

The language $\{ x \in \{0,1\}^* \mid x = reverse(x) \}$

is described by a CFG $G=(V,\Sigma,S,P)$ where

- $V = \{S\}$
- $\Sigma = \{0,1\}$
- $P = \{S \to \Lambda\ ,$
  $S \to 0,$
  $S \to 1,$
  $S \to 0S0,$
  $S \to 1S1\}$

alternative notation:
$S \to \Lambda \mid 0 \mid 1 \mid 0S0 \mid 1S1$

Why "Context-Free"?

- $\alpha_1 A \alpha_2 \Rightarrow \alpha_1 \gamma \alpha_2$ if the grammar contains the production $A \rightarrow \gamma$

- Thus, γ may substitute for A **independently of the context** (α_1 and α_2)

Algebraic Expressions

A CFG $G=(V,\Sigma,S,P)$ for **algebraic expressions**:

- $V = \{ S \}$
- $\Sigma = \{ +, -, *, /, (,), x, y, z \}$
- productions in P:

$$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$$

- `x*y+z-(z/y+x)`

Derivations

Three derivations of the string $x * y + z$:

① $S \Rightarrow S + S \Rightarrow S * S + S$
$\Rightarrow x * S + S \Rightarrow x * y + S \Rightarrow x * y + z$

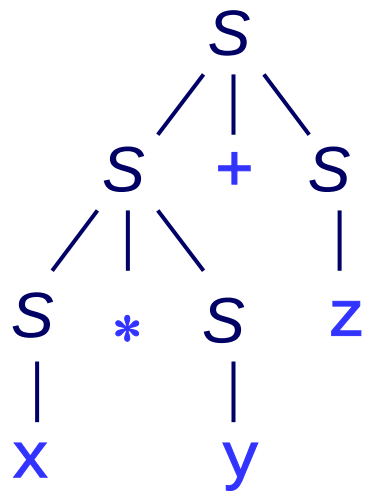
② $S \Rightarrow S + S \Rightarrow S + z$
$\Rightarrow S * S + z \Rightarrow S * y + z \Rightarrow x * y + z$

③ $S \Rightarrow S * S \Rightarrow S * S + S$
$\Rightarrow x * S + S \Rightarrow x * y + S \Rightarrow x * y + z$

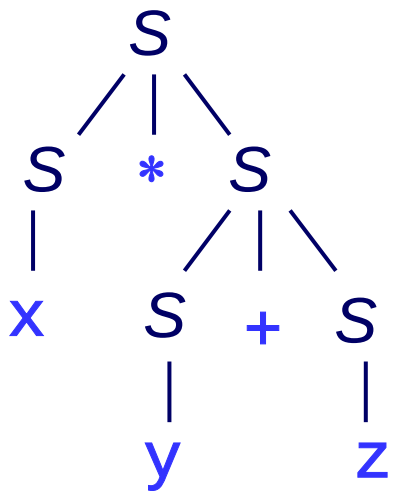
Derivation Trees

A derivation tree shows the **structure** of a derivation, but not the detailed order:

① and ②



③



A **parser** finds a derivation tree for a given string.

Ambiguous CFGs

- A CFG G is **ambiguous** if there exists a string $x \in L(G)$ with **more than one derivation tree**

- Thus, the CFG for algebraic expressions is **ambiguous**

A Simpler Syntax for Grammars

$$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$$

- V is inferred from the left-hand sides
- Σ contains the remaining symbols
- S is the first nonterminal symbol used
- P is written explicitly

Rewriting Grammars

- The ambiguous grammar:

  $S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid$
  $x \mid y \mid z$

  may be rewritten to become unambiguous:

  $S \rightarrow S + T \mid S - T \mid T$

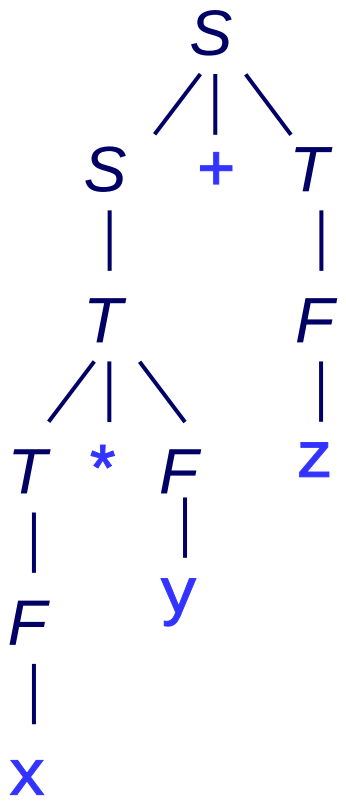
  $T \rightarrow T * F \mid T / F \mid F$

  $F \rightarrow x \mid y \mid z \mid (S)$

- This imposes an **operator precedence**

Unambiguous Parsing

- The string x * y + z now only admits a single parse tree:



Chomsky Normal Form

- A CFG $G=(V,\Sigma,S,P)$ is in **Chomsky Normal Form** (CNF) if every production in P is of the form

$$A \rightarrow BC \text{ or } A \rightarrow a$$

for $A,B,C \in V$ and $a \in \Sigma$

- Any CFG G can be rewritten to a CFG G' where G' is in Chomsky Normal Form and $L(G') = L(G) - \{\Lambda\}$
  - For every terminal a that appears in a body of length 2 or more create now production A -> a and replace a by A in the body
  - Break productions with more than two variables into group of productions with two variables
- CNF transformation preserves unambiguity

Parsing Any CNF Grammar

- Given a grammar G in CNF and a string x of length n
- Define an $|V| \times n \times n$ table P of booleans where $P(A,i,j)$ iff $A \Rightarrow^* x[i..j]$
- Using dynamic programming, fill in this table bottom-up in time $O(n^3)$ (CYK-Algorithm)
- Now, $x \in L(G)$ iff $P(S,0,n-1)$ is true

- This algorithm can easily be extended to also construct a parse tree

An Impractical Approach

- Cubic time is much too slow in practice:
  the source code for Windows is 60 million lines

- An industrial parser must be close to linear time

Shift-Reduce Parsing

- Extend the grammar with an EOF symbol *$*
- Choose between the following actions:
  - **shift**:
    move first input token to the top of a stack
  - **reduce**:
    replace α on top of the stack by *A* for some production $A \rightarrow \alpha$
  - **accept**:
    when *S$* is reduced

Shift-Reduce in Action

```
                x*y+z$ shift
x                *y+z$ reduce F->x
F                *y+z$ reduce T->F
T                *y+z$ shift
T*                y+z$ shift
T*y                 +z$ reduce F->y
T*F                 +z$ reduce T->T*F
T                   +z$ reduce S->T
S                   +z$ shift
S+                   z$ shift
S+z                    $ reduce F->z
S+F                    $ reduce T->F
S+T                    $ reduce S->S+T
S                      $ shift
S$                       accept
```

Shift-Reduce Always Works

A shift-reduce trace is the same as a backward right-most derivation sequence:

```
                x*y+z$ shift
 x               *y+z$ reduce F->x
 F               *y+z$ reduce T->F
 T               *y+z$ shift
 T*               y+z$ shift
 T*y               +z$ reduce F->y
 T*F               +z$ reduce T->T*F
 T                 +z$ reduce S->T
 S                 +z$ shift
 S+                 z$ shift
 S+z                 $ reduce F->z
 S+F                 $ reduce T->F
 S+T                 $ reduce S->S+T
 S                   $ shift
 S$                    accept
```

```
  x*y+z
⇑ F*y+z
⇑
  T*y+z
⇑ T*F+z
⇑ T+z
⇑
  S+z
⇑ S+F
⇑ S+T
⇑
  S
```

Deterministic Parsing

- A string is parsed by a grammar iff it is accepted by **some** run of the shift-reduce parser
- We must know when to shift and when to reduce

- A **deterministic** parser uses a table to determine which action to take

- Some grammars can be parsed like this

The LALR(1) Algorithm

Enumerate the productions of the grammar:

1: $S \rightarrow S + T$
2: $S \rightarrow S - T$
3: $S \rightarrow T$
4: $T \rightarrow T * F$
5: $T \rightarrow T / F$
6: $T \rightarrow F$
7: $F \rightarrow x$
8: $F \rightarrow y$
9: $F \rightarrow z$
10: $F \rightarrow (S)$

LALR(1) Setup

- A finite set of states (numbered 0, 1, 2, ...)
- An input string
- A stack of states, initially just containing state 0
- A magical table

LALR(1) Table

	x	y	z	+	-	*	/	(	)	$	S	T	F
0	s1	s2	s3					s4		a	g5	g6	g7
1	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7			
2	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8			
3	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9			
4	s1	s2	s3					s4			g8	g6	g7
5				s10	s11					s9			
6	r3	r3	r3	r3	r3	s12	s13	r3	r3	r3			
7	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6			
8				s10	s11				s14				
9	a	a	a	a	a	a	a	a	a	a			
10	s1	s2	s3					s4				g15	g7
11	s1	s2	s3					s4				g16	g7
12	s1	s2	s3					s4					g17
13	s1	s2	s3					s4					g18
14	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10			
15	r1	r1	r1	r1	r1	s12	s13	r1	r1	r1			
16	r2	r2	r2	r2	r2	s12	s13	r2	r2	r2			
17	r4	r4	r4	r4	r4	r4	r4	r4	r4	r4			
18	r5	r5	r5	r5	r5	r5	r5	r5	r5	r5			

LALR(1) Actions

- **s***k*: shift and push state *k*
- **g***k*: push state *k (”goto action”)*
- **r***i*: pop |α| states and lookup another action at place (*j*,*A*), where *j* is the new stack top and $A \rightarrow \alpha$ is the *i*'th production
  - The action at (*j*,*A*) is always a goto action
- **a**: accept
- an empty table entry indicates a parse error

LALR(1) Parsing

- Keep executing the action at place (*j*,*a*), where *j* is the stack top and *a* is the next input symbol
- Stop at either accept or error

- This is completely deterministic
- The time complexity is linear in the input string

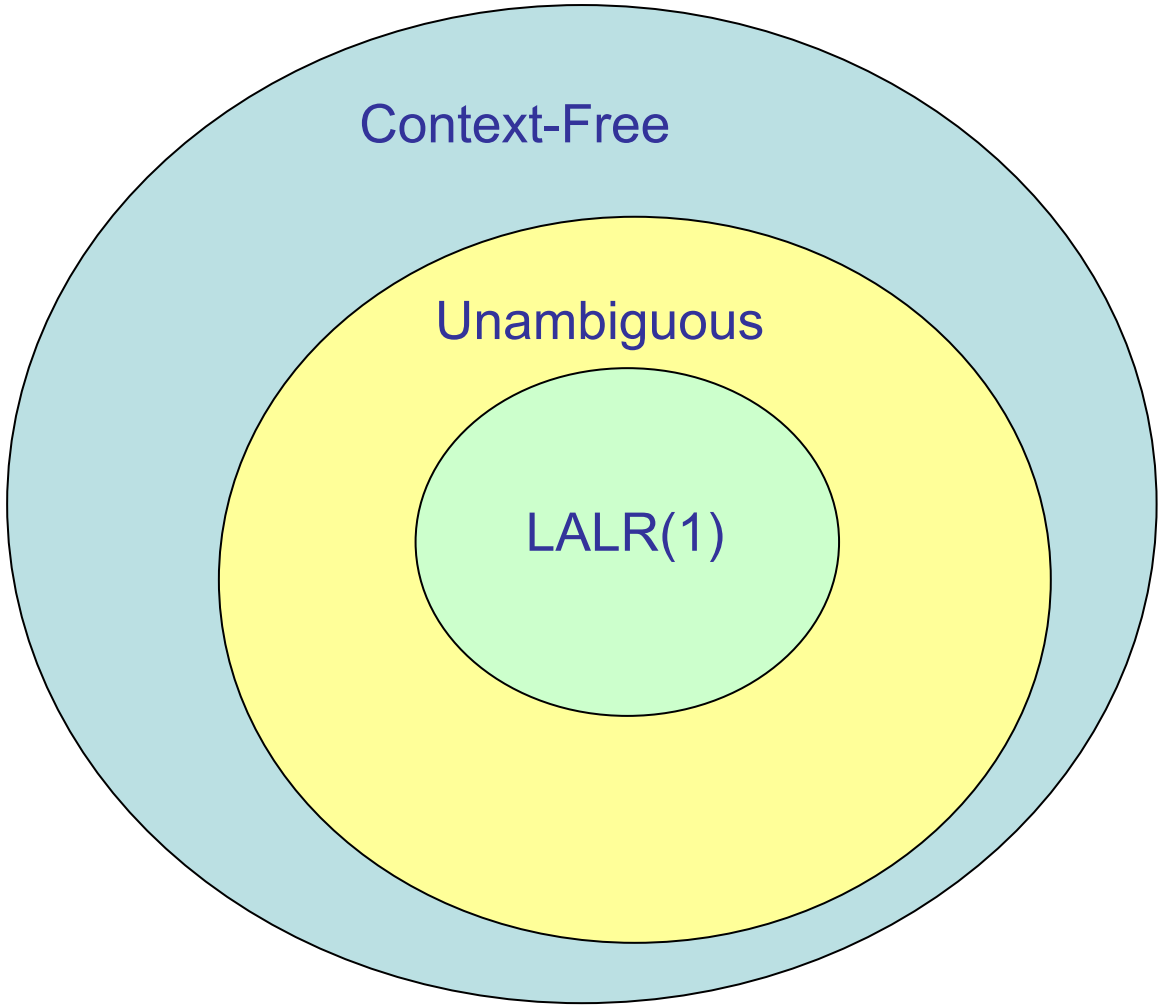
LALR(1) Example

```
0             x*y+z$   s1
0,1            *y+z$   r7
0,7            *y+z$   r6
0,6            *y+z$   s12
0,6,12          y+z$   s2
0,6,12,2          +z$  r8
0,6,12,17         +z$  r4
0,6               +z$  r3
0,5               +z$  s10
0,5,10             z$  s3
0,5,10,3            $  r9
0,5,10,7            $  r6
0,5,10,15           $  r1
0,5                 $  s9
0,5,9                  a
```

LALR(1) Conflicts

- The LALR(1) algorithm tries to construct a table
- For some grammars, the table becomes perfect
- For other grammars, it may contain conflicts:
  - **shift/reduce**:
    an entry contains both a shift and a reduce action
  - **reduce/reduce**:
    an entry contains two different reduce actions

Grammar Containments



LALR(1) Conflicts in Action

- The ambiguous grammar

$$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$$

generates 16 LALR(1) shift/reduce conflicts

- But the unambiguous version is LALR(1)...

Tokens

- For a Java grammar, Σ = Unicode
- This is not a practical approach
- Instead, grammars use an alphabet of tokens:
  - keywords
  - identifiers
  - numerals
  - strings constants
  - comments
  - symbols (==, <=, ++, ...)
  - whitespace
  - ...

Tokens Are Regular Expressions

- Tokens are defined through regular expressions:
  - keyword: `class`
  - identifier:`[a-z][a-z0-9]*`
  - numeral:`[+]?[0-9]+`
  - symbol: `++`
  - whitespace: `[ ]*`

Scanning

- A scanner translates a string of characters into a string of tokens
- It is defined by an ordered sequence of regular expressions for the tokens: $r_1, r_2, \ldots, r_k$
- Let t_i be the longest prefix of the input string that is recognized by r_i
- Let $k = max\{ |t_i| \}$
- Let $j = min\{ i \mid |t_i| = k\}$
- The next token is then t_j

Scanning with a DFA

- Scanning (for each token definition) can be efficiently performed with a minimal DFA
- Run the input string through the DFA
- At each accept state, record the current prefix
- When the DFA crashes, the last prefix is the candidate token

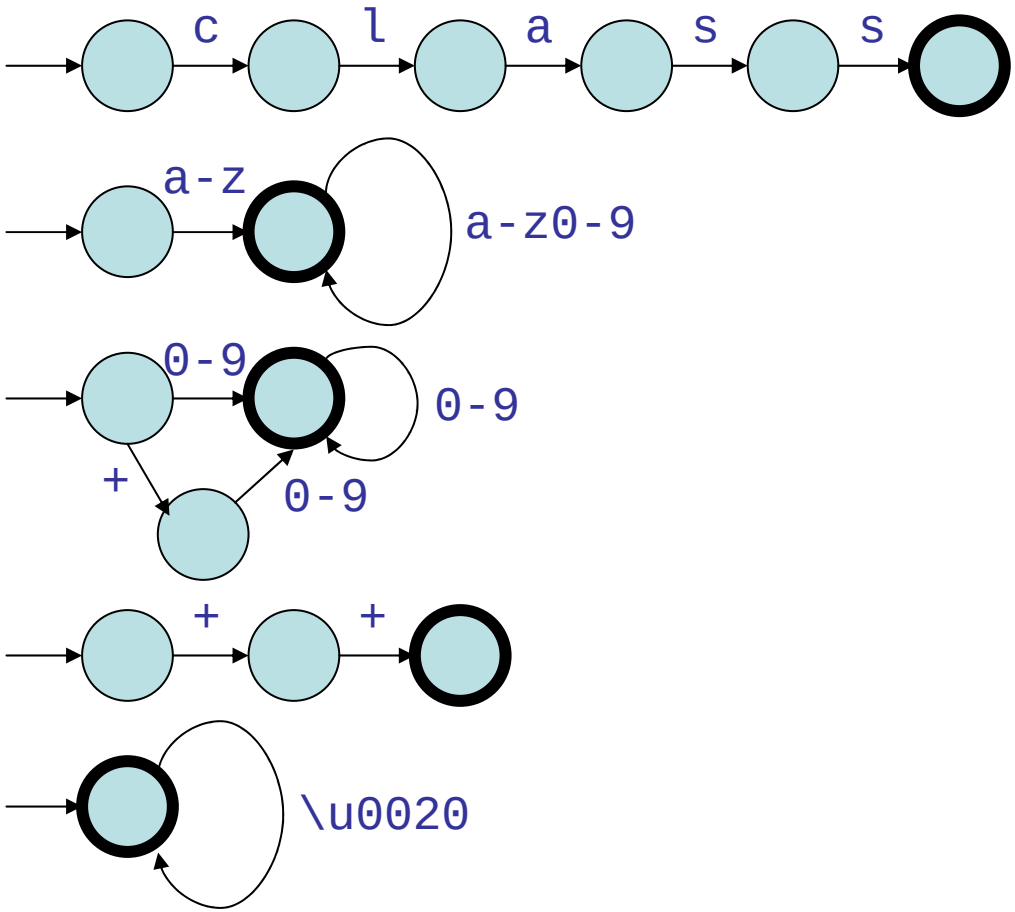
Scanning in Action (1/3)

- The previous collection of tokens:

```
class
[a-z][a-z0-9]*
[+]?[0-9]+
++
[ ]*
```

Scanning in Action (2/3)

- The automata:



Scanning in Action (3/3)

- The input string:
  `class foo +17 c++`
  generates the tokens:

  keyword: `class`
  whitespace
  identifier: `foo`
  whitespace
  numeral: `+17`
  whitespace
  identifier: `c`
  symbol: `++`