Compilers and Language Processing Tools

Summer Term 2013

Arnd Poetzsch-Heffter
Annette Bieniusa

Software Technology Group
TU Kaiserslautern

Content of Lecture

Content of Lecture (2)

5. Selected Topics in Compiler Construction
   - 5.1 Garbage Collection
   - 5.2 Just-in-time Compilation
   - 5.3 XML Processing (DOM, SAX, XSLT)

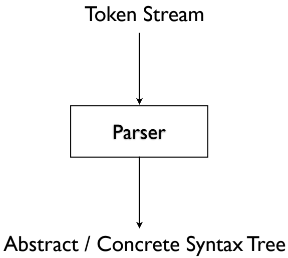
2.2 Context-Free Syntax Analysis

Section outline

1. Specification of parsers
2. Implementation of parsers
   2.1 Top-down syntax analysis
      - Recursive descent
      - LL(k) parsing theory
      - LL parser generation
   2.2 Bottom-up syntax analysis
      - Principles of LR parsing
      - LR parsing theory
      - SLR, LALR, LR(k) parsing
      - LALR parser generation
3. Error handling
4. Concrete and abstract syntax

Task of context-free syntax analysis

- Check if token stream (from scanner) matches context-free syntax of language
  - if erroneous: error handling
  - if correct: construct syntax tree



Task of context-free syntax analysis (2)

Remarks:

- Parsing can be interleaved with other actions processing the program (e.g. evaluation, adding attributes).
- Syntax tree controls the translation.
  - Concrete syntax tree corresponds closely to the context-free grammar of the language
  - Abstract syntax tree provides a more compact representation tailored to subsequent phases

2.2.1 Specification of Parsers

Specification of parsers

General specification techniques

- Context-free grammars (often in extended form)
- Syntax diagrams

Context-free grammars

Definition

Let

- N and T be two alphabets with $N \cap T = \emptyset$
- Π a finite subset of $N \times (N \cup T)^*$
- $S \in N$

Then, $\Gamma = (N, T, \Pi, S)$ is a context-free grammar (CFG) where

- N is the set of nonterminals
- T is the set of terminals
- Π is the set of productions rules
- S is the start symbol (axiom)

Context-free grammars (2)

Notations:

- $A, B, C, \ldots$ denote nonterminals
- $a, b, c, \ldots$ denote terminals
- $x, y, z, \ldots$ denote strings of terminals, i.e. $x \in T^*$
- $\alpha, \beta, \gamma, \psi, \phi, \sigma, \tau$ are strings of terminals and nonterminals, i.e. $\alpha \in (N \cup T)^*$

Productions are denoted by $A \to \alpha$.

The notation $A \to \alpha \mid \beta \mid \gamma \mid \ldots$ is an abbreviation for $A \to \alpha$, $A \to \beta$, $A \to \gamma$, $\ldots$

Derivation

Let $\Gamma = (N, T, \Pi, S)$ be a CFG:

- ψ is directly derivable from ϕ in Γ and ϕ directly produces ψ, written as $\phi \Rightarrow \psi$, if there are σ, τ with $\phi = \sigma A \tau$ and $\psi = \sigma \alpha \tau$ and $A \rightarrow \alpha \in \Pi$
- ψ is derivable from ϕ in Γ, written as $\phi \Rightarrow^* \psi$, if there exist $\phi_0, \ldots, \phi_n$ with $\phi = \phi_0$ and $\psi = \phi_n$ and $\phi_i \Rightarrow \phi_{i+1}$ for all $i \in \{0, \ldots, n-1\}$.
- $\phi_0, \ldots, \phi_n$ is called a derivation of ψ from ϕ.
- $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$.

Derivation (2)

- A derivation $\phi_0, \ldots, \phi_n$ is a leftmost derivation (rightmost) if in every derivation step $\phi_i \Rightarrow \phi_{i+1}$ the leftmost (rightmost) nonterminal in ϕ_i is replaced.
- Leftmost and rightmost derivation steps are denoted by $\phi \underset{lm}{\Longrightarrow} \psi$ and $\phi \underset{rm}{\Longrightarrow} \psi$ resp.
- The tree representation of a derivation is a syntax tree.
- $L(\Gamma) = \{z \in T^* \mid S \Rightarrow^* z\}$ is the language generated by Γ.
- $x \in L(\Gamma)$ is a sentence of Γ (germ. Satz).
- $\phi \in (N \cup T)^*$ with $S \Rightarrow^* \phi$ is a sentential form of Γ (germ. Satzform).

Derivation (3)

Remarks:

- Each derivation corresponds to exactly one syntax tree.
- For each syntax tree, there might exist several derivations.
- For “syntax tree”, the term “derivation tree” is also used.
- Several different grammars might generate the same language, i.e., the mapping

$$L : \mathit{Grammar} \rightarrow \mathit{Language}$$

is in general not injective.

Ambiguity of sentences and grammars

- A sentence is unambiguous if it has exactly one syntax tree. A sentence is ambiguous if it has more than one syntax tree.
- For each syntax tree, there exists exactly one leftmost derivation and exactly one rightmost derivation.
- Thus, a sentence is unambiguous iff it has exactly one leftmost (rightmost) derivation.
- A grammar is ambiguous if it contains an ambiguous sentence.
- For programming languages, unambiguous grammars are essential as the semantics and the translation are defined over the structure of the syntax trees.

Ambiguity of sentences and grammars (2)

Example 1: Grammar Γ_0 for expressions:

- $S \rightarrow E$
- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- $E \rightarrow ID$

Consider the input string

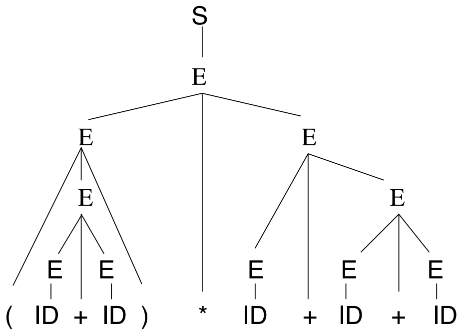
$$(av + av) * bv + cv + dv$$

resulting in the following input for the context-free analysis

$$(ID + ID) * ID + ID + ID$$

Ambiguity of sentences and grammars (3)

Syntax tree for $(ID + ID) * ID + ID + ID$



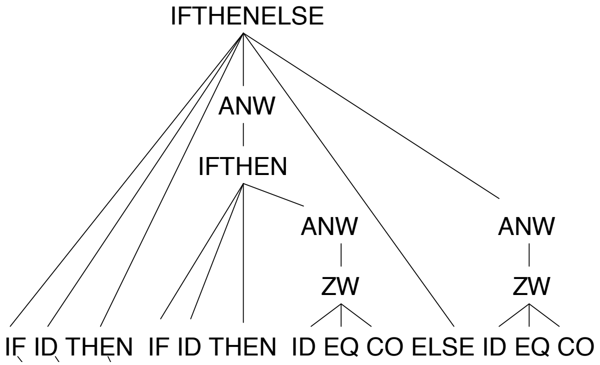
- Syntax tree does not match conventional rules of arithmetic.
- There are several syntax trees according to Γ_0 for this input, hence Γ_0 is ambiguous.

Ambiguity of sentences and grammars (4)

Example 2: Ambiguity of if-then-else construct

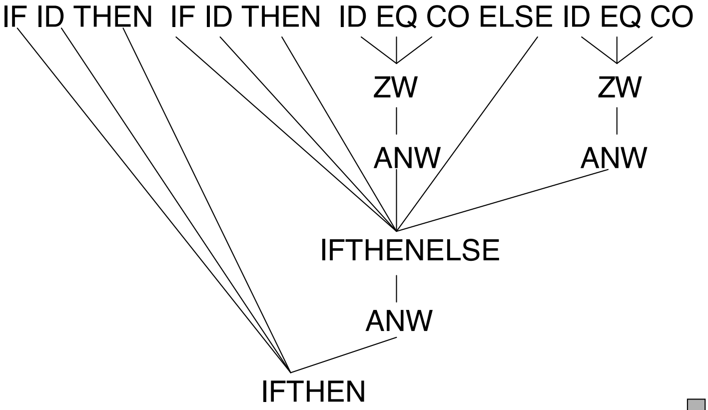
```
if B then if C then A = 9 else A = 7
```

First Derivation



Ambiguity of sentences and grammars (5)

Second Derivation

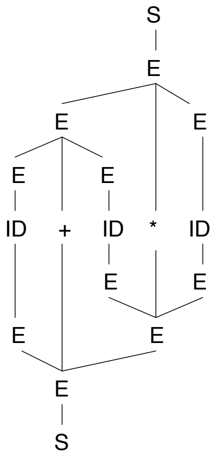


Ambiguity of grammars vs. of languages

The grammar for expressions Γ_0 is an example of an ambiguous grammar.

Γ_0:

- $S \to E$
- $E \to E + E$
- $E \to E * E$
- $E \to (E)$
- $E \to ID$

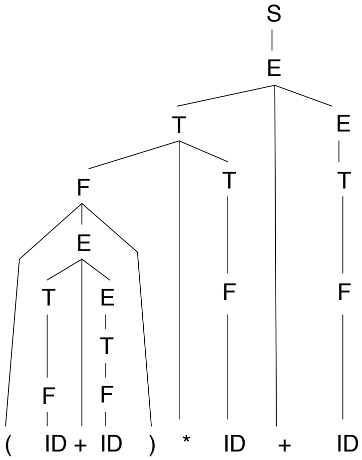


Ambiguity of grammars vs. of languages (2)

But there exists an unambiguous grammar for the same language:

Γ_1:

- $S \rightarrow E$
- $E \rightarrow T + E \mid T$
- $T \rightarrow F * T \mid F$
- $F \rightarrow (E) \mid ID$



Ambiguity of grammars vs. of languages (3)

Remark:

- A context-free language for which every grammar is ambiguous is called inherently ambiguous.
- There are inherently ambiguous CFLs.

Literature

Recommended reading:

- Wilhelm, Seidl, Hack: Band 2, Kap. 3, S. 49–60 (Syntaktische Analyse)
- Wilhelm, Maurer: Chapter 8, pp. 271–283 (Syntactic Analysis)
- Appel: Chapter 3, pp. 40–47

Parser generators

Learning objectives

- Usage of parser generators
- Characteristics of parser generators

Parser generators: Beaver

- Beaver is a LALR(1) parser generator
  `http://beaver.sourceforge.net`
- Java-based generator for grammars in EBNF
- JFlex can be used to generate cooperating scanners
- Running Beaver:

```
java -jar beaver.jar [options] language.grammar
```

- Options (selection):

-d <dir>	defines destination directory for generated files
-a	generates additional .stat file listing parser states and their actions

Structure of a Beaver specification: Headers and options (1)

```
/* Java comments that will be inserted verbatim
   at the top of the generated source file */
%header {: java code :} ;

/* Package name of the parser */
%package "package.name" ;

%import "package_or_Type" [, "package_or_Type" ...] ;

/* Java class for generated parser;
   if not used, specification file name is used */
%class "ClassName" ;
```

Structure of a Beaver specification: Headers and options (2)

```
/* Terminals that the scanner will provide */
%terminals symbol [, symbol ...] ;

/* Java types for the grammar symbols to make values availa
%typeof symbol [, symbol ...] = "JavaType" ;

/* Precedence and associativity of terminals,
  highest precedence is listed first */
%left symbol [, symbol ...] ; // (Also %right and %nonassoc

/* Symbol that is produced as a result of parsing */
%goal symbol ;
```

Example: Beaver specification for Γ_1

```
%class "ExpParser";
%terminals LPAREN, RPAREN, PLUS, TIMES, IDENTIFIER;

%goal S;

S = E;
E = T PLUS E
  | T;
T = F TIMES T
  | F;
F = LPAREN E RPAREN
  | IDENTIFIER;
```

Example: JFlex specification for Γ_1 (1)

```
import ExpParser.Terminals;
%%
// Signature for the generated scanner
%public
%final
%class ExpScanner
%extends beaver.Scanner

// Interface between the scanner and the parser is the nextToken() method
%type beaver.Symbol
%function nextToken
%yylexthrow beaver.Scanner.Exception
%eofval{
return new beaver.Symbol(Terminals.EOF, "end-of-file");
%eofval}

// store line and column information in the tokens
%line
%column
```

Example: JFlex specification for Γ_1 (2)

```
%{
 private beaver.Symbol sym(short id) {
   return new beaver.Symbol(id, yyline + 1, yycolumn + 1, yylength(), yyte
  }
%}

WhiteSpace = \r|\n|\r\n | [ \t\f]
ID = [a-z]+
%%
{WhiteSpace} { }
"(" { return sym(Terminals.LPAREN); }
")" { return sym(Terminals.RPAREN); }
"+" { return sym(Terminals.PLUS); }
"*" { return sym(Terminals.TIMES); }
{ID} { return sym(Terminals.IDENTIFIER); }

// fall through errors
. { throw new beaver.Scanner.Exception("illegal character \"" +
            yytext() + "\" at line " + yyline + "," + yycolumn); }
```

Structure of generated parser code

- Output files `ExpParser.java` and `ExpScanner.java`
- Table for LALR automaton `ExpParser.stat` (parser option –a)
  - Indicates what action (shift, reduce, or error) is to be taken on each lookahead symbol when encountered in each state
  - Shows which state to shift to after reduction

Usage of generated parser

- Parser calls scanner with `nextToken()` method when a new terminal token is needed
- Initialising and running parser with scanner

```
ExpParser parser = new ExpParser();
Object result = parser.parse(new ExpScanner(
   new FileReader("input.txt")))
```

- Actual type of `result` is given by type definition, e.g. `%typeof E = "String"`

2.2.2 Implementation of Parsers

Implementation of parsers

Overview

- Top-down parsing
  - Recursive descent
  - LL parsing
  - LL parser generation
- Bottom-up parsing
  - LR parsing
  - LALR, SLR, LR(k) parsing
  - LALR parser generation

Methods for context-free analysis

- Manually developed, grammar-specific implementation (error-prone, inflexible, possibly more efficient)
- Backtracking (simple, but maybe inefficient)
- Cocke-Younger-Kasami-Algorithm (1967):
  - for all CFGs in Chomsky normal form
  - based on idea of dynamic programming
  - time complexity $O(n^3)$ (however, often linear complexity desired)
- Top-down methods: from axiom to word/token stream
- Bottom-up methods: from word/token stream to axiom

Example: Top-down analysis

Top-down analysis from left to right leads to leftmost derivation.
Example derivation with Γ_1:

							S		$\Rightarrow$
							E		$\Rightarrow$
						T	+	E	$\Rightarrow$
		F			*	T	+	E	$\Rightarrow$
(		E		)	*	T	+	E	$\Rightarrow$
(	T	+	E	)	*	T	+	E	$\Rightarrow$
(	F	+	E	)	*	T	+	E	$\Rightarrow$
(	ID	+	E	)	*	T	+	E	$\Rightarrow$
(	ID	+	T	)	*	T	+	E	$\Rightarrow$
(	ID	+	F	)	*	T	+	E	$\Rightarrow$
(	ID	+	ID	)	*	T	+	E	$\Rightarrow$
(	ID	+	ID	)	*	F	+	E	$\Rightarrow$
(	ID	+	ID	)	*	ID	+	E	$\Rightarrow$
(	ID	+	ID	)	*	ID	+	T	$\Rightarrow$
(	ID	+	ID	)	*	ID	+	F	$\Rightarrow$
(	ID	+	ID	)	*	ID	+	ID	

Example: Bottom-up analysis

Bottom-up analysis from left to right leads to rightmost derivation.
Example derivation with Γ_1:

(	ID	+	ID	)	*	ID	+	ID	⇐
(	F	+	ID	)	*	ID	+	ID	⇐
(	T	+	ID	)	*	ID	+	ID	⇐
(	T	+	F	)	*	ID	+	ID	⇐
(	T	+	T	)	*	ID	+	ID	⇐
(	T	+	E	)	*	ID	+	ID	⇐
(		E		)	*	ID	+	ID	⇐
			F		*	ID	+	ID	⇐
			F		*	F	+	ID	⇐
			F		*	T	+	ID	⇐
						T	+	ID	⇐
						T	+	F	⇐
						T	+	T	⇐
						T	+	E	⇐
							E		⇐
							S		

Context-free analysis with linear complexity

- Restrictions on grammar (not every CFG has a linear parser)
- Use of push-down automata or systems of recursive procedures
- Solve the possible conflicts by <u>lookahead</u> into the input rest

Syntax analysis methods and parser generators

- Basic knowledge of syntax analysis is essential for use of parser generators.
- Parser generators are not always applicable.
- Often, error handling has to be done manually.
- Methods underlying parser generation is a good example for a generic technique (and a highlight of computer science!).

2.2.2.1 Top-down syntax analysis

Top-down syntax analysis

Learning objectives

- Understand the general principle of top-down syntax analysis
- Be able to implement recursive descent parsing (by example)
- Know expressiveness and limitations of top-down parsing
- Understand the basic concepts of LL(k) parsing

Recursive descent parsing

Basic idea

- Each nonterminal A is associated with a procedure.
  This procedure accepts a partial sentence derived from A.
- The procedure implements a finite automaton constructed from the productions with A as left-hand side. This automaton is called the item automaton of A.
- The recursiveness of the grammar is mapped to mutual recursive procedures such that the stack of higher programing languages is used for handling the recursion.

Construction of recursive descent parser

Let Γ_1' be a CFG accepting $w\#$ iff $w \in L(\Gamma_1)$, where # is used as a special character denoting the end of the input.

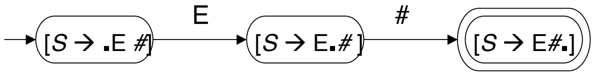
Γ_1':

- $S \rightarrow E\#$
- $E \rightarrow T + E \mid T$
- $T \rightarrow F * T \mid F$
- $F \rightarrow (E) \mid ID$

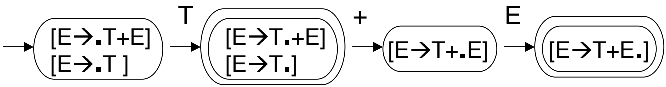
Construct an item automaton for each nonterminal.

Item automata

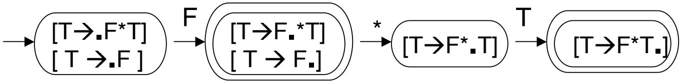
$S \to E\#$



$E \to T + E \mid T$

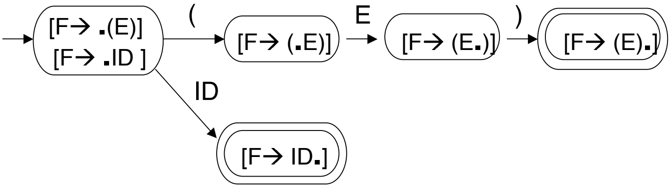


$T \to F * T \mid F$



Item automata (2)

$F \rightarrow (E) \mid ID$



Recursive descent parsing procedures

- The recursive procedures are then constructed from the item automata.
- The input is a token stream terminated by #.
- The variable `currToken` contains one token lookahead, i.e., the first symbol of the input rest.

Recursive descent parsing procedures (2)

Production: $S \rightarrow E\#$

```
void S() {
   E();
   if( currToken == '#' ) {
     accept();
   } else {
     error();
   }
}
```

Recursive descent parsing procedures (3)

Production: $E \rightarrow T + E \mid T$

```
void E() {
   T();
   if( currToken == '+' ) {
      readToken();
      E();
   }
}
```

Production: $T \rightarrow F * T \mid F$

```
void T() {
   F();
   if( currToken == '*' ){
      readToken();
      T();
   }
}
```

Recursive descent parsing procedures (4)

Production: $F \to (\ E\ )\ |\ ID$

```
void F() {
   if( currToken == '(' ) {
     readToken();
     E();
     if( currToken == ')' ) {
       readToken();
     } else error();
   } else if( currToken == ID ) {
     readToken();
   } else
     error();
}
```

Recursive descent parsing procedures (5)

Remarks:

- Recursive descent
  - is relatively easy to implement
  - can easily be combined with other tasks (see following example)
  - is a typical example for syntax-directed methods (see also following example)
- Example above uses one token lookahead.
- Error handling is not considered.

Recursive descent and evaluation

Example: Interpreter for expressions using recursive descent with an environment for the variables appearing in the expression:

```
int env(Ident); // Ident -> int
```

The intermediate evaluation results are stored in local variables `imr`:

```
int S() {
  int imr = E();
  if (currToken == '#') {
    return imr;
  } else {
    error();
  }
}
```

Recursive descent and evaluation (2)

```
int E() {
  int imr = T();
  if( currToken == '+' ) {
    readToken();
    imr = imr + E();
  }
  return imr;
}

int T() {
  int imr = F();
  if (currToken == '*'){
    readToken();
    imr = imr * T();
  }
  return imr;
}
```

Recursive descent and evaluation (3)

```
int F() {
  if (currToken == '('){
    readToken();
    int imr = E();
    if (currToken == ')'){
      readToken();
      return imr;
    } else {
      error();
    }
  } else if (currToken == ID) {
    readToken();
    return env(code(ID));
  } else {
    error();
  }
}
```

Recursive descent and evaluation (4)

- Extension of parser with actions/computations
  - is relatively simple to implement
  - mixes conceptually different phases/tasks
  - may lead to unmaintainable programs
- Question: For which grammars does the recursive descent technique work?
  $\rightarrow$ **LL(k) parsing theory**

LL parsing

- Basis for town-down syntax analysis
- First “L” refers to reading input from **left** to right
- Second “L” refers to search for **leftmost** derivations

LL(k) grammars

Definition (LL(k) grammar)

Let $\Gamma = (N, T, \Pi, S)$ be a CFG and $k \in \mathbb{N}$.

Γ is an LL(k) grammar if for any two leftmost derivations

$$S \underset{lm}{\Longrightarrow}^* uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \underset{lm}{\Longrightarrow}^* ux$$

and

$$S \underset{lm}{\Longrightarrow}^* uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \underset{lm}{\Longrightarrow}^* uy$$

the following holds:

$$\text{If } prefix(k, x) = prefix(k, y), \text{ then } \beta = \gamma$$

where $prefix(k, x)$ yields the longest prefix of x with length $\leq k$.

LL(k) grammars (2)

Definition (LL(k) language)

A language $L_k \subseteq \Sigma^*$ is LL(k) if there exists an LL(k) grammar Γ with $L(\Gamma) = L_k$.

Remarks:

- A grammar is an LL(k) grammar if for a leftmost derivation with k token lookahead the correct production for the next derivation step can be found.
- The definition of LL(k) grammars provides no method to test if a grammar has the LL(k) property.

Non LL(k) grammars

Example 1: Grammar with left recursion Γ_2:

- $S \rightarrow E\#$
- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid ID$

Elimination of left recursion:
Replace productions of form $A \rightarrow A\alpha \mid \beta$ where β does not start with A by $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \epsilon$.

Non LL(k) grammars (2)

Elimination of left recursion: From Γ_2 we obtain Γ_3.

Γ_2:

- $S \rightarrow E\#$
- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid ID$

Γ_3

- $S \rightarrow E\#$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT \mid \epsilon$
- $F \rightarrow (E) \mid ID$

Non LL(k) grammars (3)

Example 2: Grammar Γ_4 with unlimited lookahead

- $STM \rightarrow VAR := VAR \mid ID(IDLIST)$
- $VAR \rightarrow ID \mid ID(IDLIST)$
- $IDLIST \rightarrow ID \mid ID, IDLIST$

Γ_4 is not an LL(k) grammar for any k.
(Proof: cf. Wilhelm, Maurer, Example 8.3.4, p. 319)

Transformation to LL(2) grammar Γ_4':

- $STM \rightarrow ASS\_CALL \mid ID := VAR$
- $ASS\_CALL \rightarrow ID(IDLIST)\ ASS\_CALL\_REST$
- $ASS\_CALL\_REST \rightarrow := VAR \mid \epsilon$

Non LL(k) grammars (4)

Remarks:

- The transformed grammars accept the same language, but generate other syntax trees:
  - From a theoretical point of view, this is acceptable.
  - From a programming language implementation perspective, this is in general not acceptable.
- There are languages L that are not LL(k) even after left-recursion elimination (e.g., grammar Γ_5 below)

Non LL(k) grammars (5)

Example 3:

For the following grammar Γ_5, there is no k such that Γ_5 is an LL(k).

- $S \to A \mid B$
- $A \to aAb \mid 0$
- $B \to aBbb \mid 1$

Remark:

For $L(\Gamma_5)$, there exists no LL(k) grammar.

Non LL(k) grammars (6)

Proof.

Let k be arbitrary, but fixed.
Choose two derivations according to the LL(k) definition and show that, despite of equal prefixes of length k, β and γ are not equal:

$$S \underset{lm}{\Longrightarrow}^* S \underset{lm}{\Longrightarrow} A \underset{lm}{\Longrightarrow}^* a^k 0 b^k$$
$$S \underset{lm}{\Longrightarrow}^* S \underset{lm}{\Longrightarrow} B \underset{lm}{\Longrightarrow}^* a^k 1 b^{2k}$$

Then, $prefix(k, a^k 0 b^k) = a^k = prefix(k, a^k 1 b^{2k})$, but
$\beta = A \neq B = \gamma$. $\square$

FIRST and FOLLOW sets

Definition

Let $\Gamma = (N, T, \Pi, S)$ be a CFG, $k \in \mathbb{N}$;

$$T^{\leq k} = \{u \in T^* \mid \mathit{length}(u) \leq k\}$$

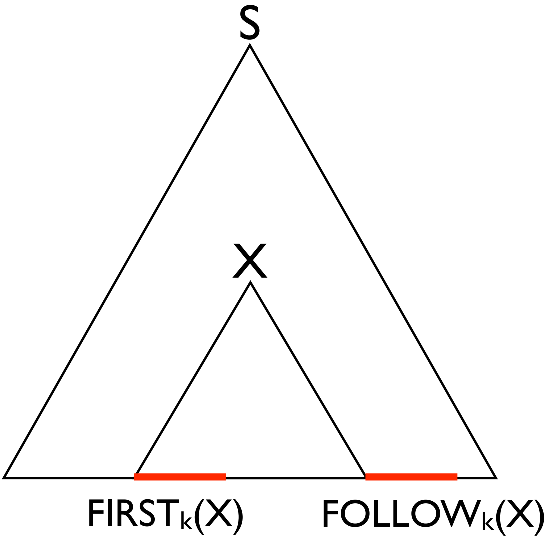
denotes the set of all prefixes of length at least k.

We define:

- $\mathit{FIRST}_k : (N \cup T)^* \rightarrow \mathcal{P}(T^{\leq k})$
  $\mathit{FIRST}_k(\alpha) = \{\mathit{prefix}(k, u) \mid \alpha \Rightarrow^* u\}$

  where $\mathit{prefix}(n, u) = u$ for all u with $\mathit{length}(u) \leq n$.
- $\mathit{FOLLOW}_k : (N \cup T)^* \rightarrow \mathcal{P}(T^{\leq k})$
  $\mathit{FOLLOW}_k(\alpha) = \{w \mid S \Rightarrow^* \beta\alpha\gamma \wedge w \in \mathit{FIRST}_k(\gamma)\}$

FIRST and FOLLOW sets in parse trees



Characterization of LL(1) grammars

Definition (reduced CFG)

A CFG $\Gamma = (N, T, \Pi, S)$ is reduced if each nonterminal occurs in a derivation and each nonterminal derives at least one word.

Lemma

A reduced CFG is LL(1) iff for any two productions $A \to \beta$ and $A \to \gamma$ the following holds:

$$\Big(FIRST_1(\beta) \oplus_1 FOLLOW_1(A)\Big) \cap \Big(FIRST_1(\gamma) \oplus_1 FOLLOW_1(A)\Big) = \emptyset$$

where $L_1 \oplus_1 L_2 = \{prefix(1, vw) \mid v \in L_1, w \in L_2\}$

Remark: FIRST and FOLLOW sets are computable, so this criterion can be checked automatically.

Example: $FIRST_k$ and $FOLLOW_k$

Check that the modified expression grammar Γ_3 is LL(1).

- $S \to E\#$
- $E \to TE'$
- $E' \to +TE' \mid \epsilon$
- $T \to FT'$
- $T' \to *FT \mid \epsilon$
- $F \to (E) \mid ID$

Compute $FIRST_1$ and $FOLLOW_1$ for each nonterminal (cf. lecture).

Example: $FIRST_k$ and $FOLLOW_k$

Apply the characterization lemma!

- $F \rightarrow (E) \mid ID$:
  $$FIRST_1((E)) \oplus_1 FOLLOW_1(F) \cap FIRST_1(ID) \oplus_1 FOLLOW_1(F)$$
  $$= \{(\} \oplus_1 FOLLOW_1(F) \cap \{ID\} \oplus_1 FOLLOW_1(F)$$
  $$= \emptyset$$
- $E' \rightarrow +TE' \mid \epsilon$:
  $$FIRST_1(+TE') \oplus_1 FOLLOW_1(E') \cap FIRST_1(\epsilon) \oplus_1 FOLLOW_1(E')$$
  $$= \{+\} \oplus_1 FOLLOW_1(E') \cap \{\epsilon\} \oplus_1 FOLLOW_1(E')$$
  $$= \{+\} \cap \{\#,)\}$$
  $$= \emptyset$$
- $T' \rightarrow *FT \mid \epsilon$:
  $$FIRST_1(*FT') \oplus_1 FOLLOW_1(T') \cap FIRST_1(\epsilon) \oplus FOLLOW_1(T')$$
  $$= \{*\} \oplus_1 FOLLOW_1(T') \cap \{\epsilon\} \oplus_1 FOLLOW_1(T')$$
  $$= \{*\} \cap \{+, \#,)\}$$
  $$= \emptyset$$

Similarly for E and T.

Proof of LL characterization lemma

- **Direction from left to right:**
  Γ is LL(1) implies FIRST-FOLLOW disjointness.
  We show: "FIRST-FOLLOW intersection non-empty" implies "not LL(1)".

  Proof by contradiction:
  Let $A \to \beta$ and $A \to \gamma$ be two distinct productions of Γ with $\beta \neq \gamma$ such that the FIRST-FOLLOW intersection is non-empty.

  We consider three cases:

  Case 1: $\beta \Rightarrow^* \epsilon$ and $\gamma \Rightarrow^* \epsilon$
  In this case, the LL(1) property does not hold for $A \to \beta$, $A \to \gamma$.

Proof of LL characterization lemma (2)

Case 2: $\beta \not\Rightarrow^* \epsilon$

Then, there is a z with $\mathit{length}(z) = 1$ and

$$z \in \Big(\mathit{FIRST}_1(\beta) \oplus_1 \mathit{FOLLOW}_1(A)\Big) \cap \Big(\mathit{FIRST}_1(\gamma) \oplus_1 \mathit{FOLLOW}_1(A)\Big)$$

Because Γ is reduced, there are two derivations:

$$S \Rightarrow^* \psi A \alpha \Rightarrow \psi \beta \alpha \Rightarrow^* \psi z x$$
$$S \Rightarrow^* \psi A \alpha \Rightarrow \psi \gamma \alpha \Rightarrow^* \psi z y$$

and there is a u such that $\psi \Rightarrow^* u$, i.e., there are leftmost derivations

$$S \underset{lm}{\Longrightarrow}^* uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \underset{lm}{\Longrightarrow}^* uzx$$
$$S \underset{lm}{\Longrightarrow}^* uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \underset{lm}{\Longrightarrow}^* uzy$$

But $\mathit{prefix}(1, zx) = z = \mathit{prefix}(1, zy)$ for $\beta \neq \gamma$ contradicts the LL(1) property of Γ.

Case 3: $\gamma \not\Rightarrow^* \epsilon$: similar to Case 2.

Proof of LL characterization lemma (3)

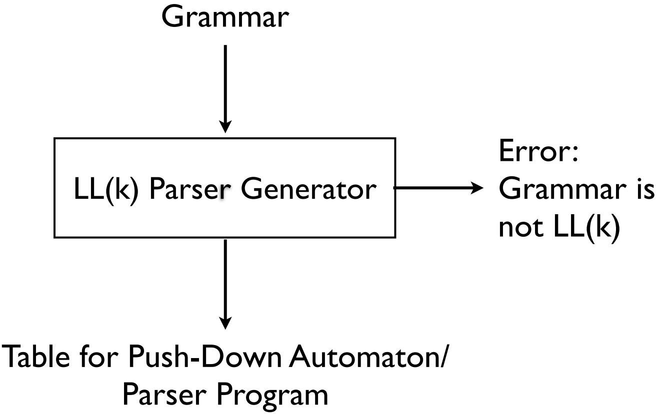
- **Direction from right to left**:
  FIRST-FOLLOW disjointness implies Γ is LL(1):

  Proof:
  Consider any two derivations with $\beta \neq \gamma$:

$$S \underset{lm}{\Longrightarrow}^* uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \underset{lm}{\Longrightarrow}^* ux$$
$$S \underset{lm}{\Longrightarrow}^* uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \underset{lm}{\Longrightarrow}^* uy$$

  Then, $prefix(1, x) \in \left(FIRST_1(\beta) \oplus_1 FOLLOW_1(A)\right)$ and $prefix(1, y) \in \left(FIRST_1(\gamma) \oplus_1 FOLLOW_1(A)\right)$.
  Because of FIRST-FOLLOW disjointness,
  $prefix(1, x) \neq prefix(1, y)$

Parser generation for LL(k) languages



Parser generation for LL(k) languages (2)

Remarks:

- Use of push-down automata with lookahead
- Select production from tables
- Advantages over bottom-up techniques in error analysis and error handling

Example system: ANTLR (`http://www.antlr.org/`)

Recommended reading for top-down analysis:

- Wilhelm, Maurer: Chapter 8, Sections 8.3.1. to Sections 8.3.4, pp. 312 - 329

2.2.2.2 Bottom-up syntax analysis

Bottom-up syntax analysis

Learning objectives:

- General principles of bottom-up syntax analysis
- LR(k) analysis
- Resolving conflicts in parser generation
- Connection between CFGs and push-down automata

Basic ideas: bottom-up syntax analysis

- Bottom-up analysis is more powerful than top-down analysis, since production is chosen at the end of the analysis while in top-down analysis the production is selected up front.
- LR: read input from left (L)
  and search for rightmost derivations (R)

Principles of LR parsing

1. Reduce from sentence to axiom according to productions of Γ
2. Reduction yields sentential forms αx with $\alpha \in (N \cup T)^*$ and $x \in T^*$ where x is the input rest
3. α has to be a prefix of a right sentential form of Γ. Such prefixes are called viable prefixes. This prefix property has to hold invariantly during LR parsing to avoid dead ends.
4. Reductions are always made at the leftmost possible position.

More precisely:

Viable prefix

Definition

Let $S \underset{rm}{\Longrightarrow}^* \beta A u \underset{rm}{\Longrightarrow} \beta\alpha u$ be a right sentential form of Γ.

Then α is called a handle or redex of the right sentential form $\beta\alpha u$.

Each prefix of $\beta\alpha$ is a viable prefix of Γ.

Regularity of viable prefixes

Theorem

The language of viable prefixes of a grammar Γ is regular.

Proof.

Cf. Wilhelm, Maurer Thm. 8.4.1 and Corrollary 8.4.2.1. (pp. 361, 362). Essential proof steps are illustrated in the following by the construction of the LR-DFA(Γ). □

Examples: towards LR parsing

- Consider Γ_1
  - $S \rightarrow aCD$
  - $C \rightarrow b$
  - $D \rightarrow a \mid b$

  Analysis of `aba` can lead to a dead end (cf. lecture).

  Considering viable prefixes can avoid this.

Examples: towards LR parsing (2)

- Consider Γ_2
  - $S \rightarrow E\#$
  - $E \rightarrow a \mid (E) \mid EE$

  Analysis of `((a))(a)#` (cf. lecture)

  Stack can manage prefixes already read.

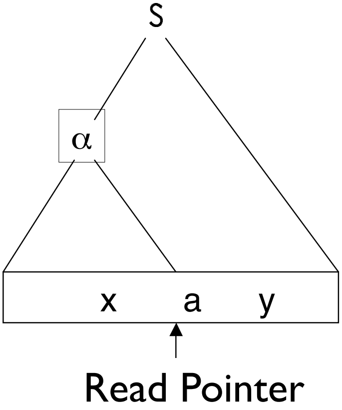
Examples: towards LR parsing (3)

- Consider Γ_3
  - $S \to E\#$
  - $E \to E + T \mid T$
  - $T \to ID$

  Analysis of `ID + ID + ID #` (cf. lecture)

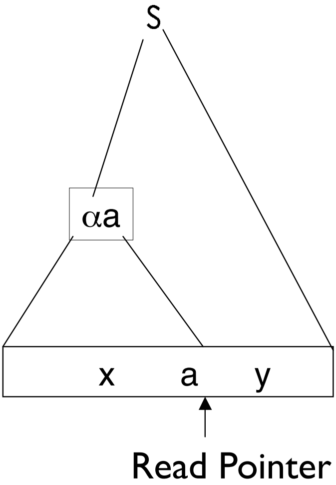
LR parsing: shift and reduce actions

Schematic syntax tree for input *xay* with
$\alpha \in (N \cup T)^*$, $a \in T$, $x, y \in T^*$ and start symbol S:

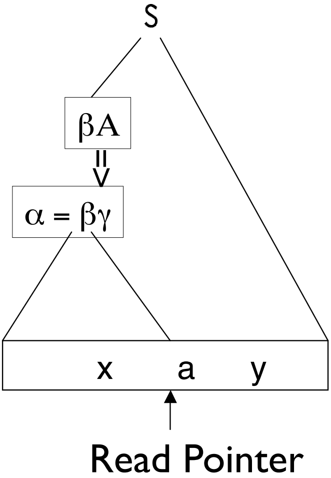


LR parsing: shift and reduce actions (2)

Shift step:



Reduce step:



LR parsing: shift and reduce actions (3)

Problems:

- Make sure that all reductions guarantee that the resulting prefix remains a viable prefix.
- When to shift? When to reduce? Which production to use?

Solution:
For each grammar Γ construct LR-DFA(Γ) automaton (also called LR(0) automaton), that describes the viable prefixes.

Construction of LR-DFA

Let $\Gamma = (T, N, \Pi, S)$ be a CFG.

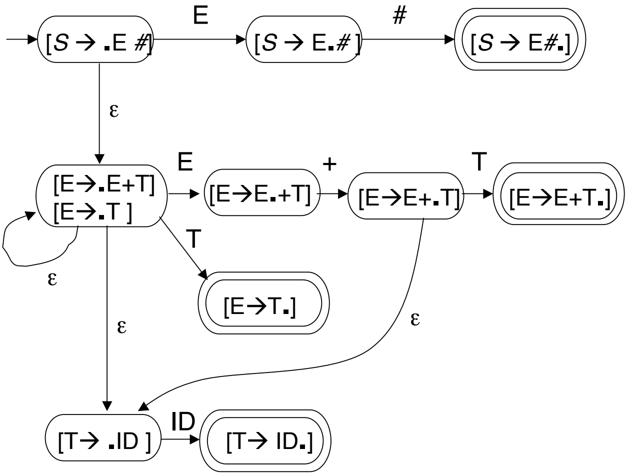
- For each nonterminal $A \in N$, construct item automaton
- Build union of item automata: Start state is the start state of item automaton for S, final states are final states of item automata
- Add ϵ transitions from each state which contains the dot in front of a nonterminal A to the starting state of the item automaton of A

Theorem

The automaton obtained from LR-DFA(Γ) by declaring all states to be final states exactly accepts the language of viable prefixes of Γ.

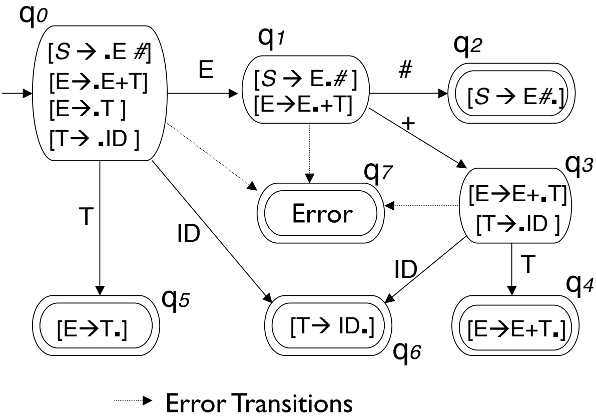
Example: Construction of LR-DFA

Γ_3: $S \to E\#,\ E \to E + T \mid T,\ T \to ID$



Example: Construction of LR-DFA (2)

Power set construction:



Viable prefixes of maximal length: $E\#$, T, ID, $E + ID$, $E + T$

Example: Construction of LR-DFA (3)

Remarks:

- In the example, each final state contains one completely read production, this is in general not the case.
- If a final state contains more than one completely read productions, we have a reduce/reduce conflict.
- If a final state contains a completely read and an incompletely read production with a terminal after the dot, we have a shift/reduce conflict.

Analysis with LR-DFA

Analysis of `ID + ID + ID #` with LR-DFA
(the viable prefix is underlined)

$$\underline{\mathrm{ID}} + \mathrm{ID} + \mathrm{ID}\ \# \qquad <=$$
$$\underline{\mathrm{T}} + \mathrm{ID} + \mathrm{ID}\ \# \qquad <=$$
$$\underline{\mathrm{E} + \mathrm{ID}} + \mathrm{ID}\ \# \qquad <=$$
$$\underline{\mathrm{E} + \mathrm{T}} + \mathrm{ID}\ \# \qquad <=$$
$$\underline{\mathrm{E} + \mathrm{ID}}\ \# \qquad <=$$
$$\underline{\mathrm{E} + \mathrm{T}}\ \# \qquad <=$$
$$\underline{\mathrm{E}\ \#} \qquad <=$$
$$\mathrm{S}$$

Analysis with LR-DFA (2)

Note:

- The sentential forms always consist of a viable prefix and an input rest.
- If an LR-DFA is used, after each reduction the sentential form has to be read from the beginning.

  Thus: Use pushdown automaton for analysis.

LR pushdown automaton

Definition

Let $\Gamma = (N, T, \Pi, S)$ be a CFG. The LR-DFA pushdown automaton for Γ contains:

- a finite set of states Q (the states of the LR-DFA(Γ))
- a set of actions $Act = \{shift, accept, error\} \cup red(\Pi)$, where $red(\Pi)$ contains an action $reduce(A \to \alpha)$ for each $A \to \alpha$.
- an action table $at : Q \to Act$.
- a successor table $succ : P \times (N \cup T) \to Q$ with $P = \{q \in Q \mid at(q) = shift\}$

LR pushdown automaton (2)

Remarks:

- The LR-DFA pushdown automaton is a variant of pushdown automata particularly designed for LR parsing.
- States encode the read left context.
- If there are no conflicts, the action table can be directly constructed from the LR-DFA:
  - accept: final state of item automaton of start symbol
  - reduce: all other final states
  - error: error state
  - shift: all other states

Execution of Pushdown Automaton

- Configuration: $Q^* \times T^*$ where variable `stack` denotes the sequence of states and variable `inr` denotes the input rest
- Start configuration: $(q_0, input)$, where q_0 is the start state of the LR-DFA
- Interpretation Procedure:

```
(stack, inr) := (q0,input);
do {
    step(stack,inr);
} while( at( top(stack) ) != accept
      && at( top(stack) ) != error );
if( at( top(stack) ) == error ) return error;
```

with

Execution of Push-Down Automaton (2)

```
void step (var StateSeq stack, var TokenSeq inr) {
  State tk: = top(stack);
  switch( at(tk) ) {
  case shift:
     stack := push ( succ(tk,top(inr)), stack );
     inr := tail(inr);
     break;
  case reduce A -> a:
     stack := mpop( length(a), stack );
     stack := push( succ(top(stack),A), stack);
     break;
  }
}
```

Example: LR push down automaton

LR-DFA with states $q_0, \ldots, q_7$ for grammar Γ_3

Action table

q0	shift	
q1	shift	
q2	accept	
q3	shift	
q4	reduce	E→E+T
q5	reduce	E→T
q6	reduce	T→ID
q7	error	

Successor table

	ID	+	#	E	T
q0	q6	q7	q7	q1	q5
q1	q7	q3	q2	q7	q7
q2					
q3	q6	q7	q7	q7	q4
q4					
q5					
q6					
q7					

Example: LR push down automaton (2)

Computation for input `ID + ID + ID #`

Stack	Input Rest	Action	
q0	ID + ID + ID #	shift	
q0 q6	+ ID + ID #	reduce	T→ID
q0 q5	+ ID + ID #	reduce	E→T
q0 q1	+ ID + ID #	shift	
q0 q1 q3	ID + ID #	shift	
q0 q1 q3 q6	+ ID #	reduce	T→ID
q0 q1 q3 q4	+ ID #	reduce	E→ E+T
q0 q1	+ ID #	shift	
q0 q1 q3	ID #	shift	
q0 q1 q3 q6	#	reduce	T→ID
q0 q1 q3 q4	#	reduce	E→ E+T
q0 q1	#	shift	
q0 q1 q2		accept	

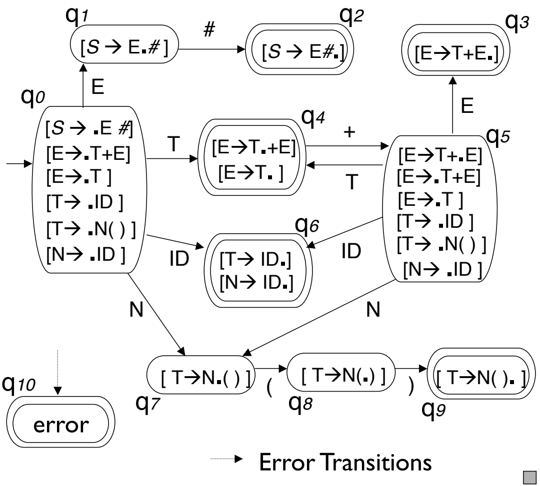
LR-DFA construction

Questions:

- Does LR-DFA construction work for all unambiguous grammars?
- For which grammars does the construction work?
- How can the construction be generalized / made more expressive?

Example: LR-DFA with conflicts

LR-DFA for Γ_6: $S \to E\#,\ E \to T + E \mid T,\ T \to ID \mid N(),\ N \to ID$



LR parsing conflicts

2 kinds of conflicts:

- shift/reduce conflicts (q_4 in example)
- reduce/reduce conflicts (q_6 in example)

LR parsing theory

Definition

Let $\Gamma = (N, T, \Pi, S)$ be a CFG and $k \in \mathbb{N}$.
Γ is an LR(k) grammar if for any two rightmost derivations

$$S \Rightarrow^*_{rm} \alpha A u \Rightarrow_{rm} \alpha\beta u$$
$$S \Rightarrow^*_{rm} \gamma B v \Rightarrow_{rm} \alpha\beta w$$

it holds that:
If $prefix(k, u) = prefix(k, w)$ then $\alpha = \gamma$, $A = B$ and $v = w$

LR parsing theory (2)

Remarks:

- While for LL grammars the selection of the production depends on the nonterminal to be derived, for LR grammars it depends on the complete left context.
- For LL grammars, the lookahead considers the language to be generated from the nonterminal. For LR grammars, the lookahead considers the language generated from not yet read nonterminals.

Characterization of LR(0)

Theorem

Let Γ be a reduced CFG;
Γ is LR(0) if-and-only-if LR-DFA(Γ) contains no conflicts.

We first present some properties of LR-DFA construction that are needed for the proof.

Properties of LR-DFA Construction

Lemma 1

(a) Each path leading to a state q with item $[A \rightarrow \alpha.\beta]$ ends with α.

(b) If q is a state with items $[A \rightarrow \alpha.\delta]$ and $[B \rightarrow \beta.\gamma]$, then α is a postfix of β or β a postfix of α.

(c) If there is a transition marked with H from state p to state q, then p contains an item of form $[C \rightarrow \gamma.H\delta]$.

Proof.

The properties

- are obvious for LR-NFA (by construction) and
- remain valid for LR-DFA.

□

Properties of LR-DFA construction (2)

Lemma 2

For any A, ψ, α, β:
Item $[A \to \alpha.\beta]$ is contained in a state that is reached by $\psi\alpha$ iff there exists a rightmost derivation

$$S \underset{rm}{\Longrightarrow}^* \psi A u \underset{rm}{\Longrightarrow} \psi\alpha\beta u$$

Proof.
not worked out (cf. Wilhelm, Maurer, Thm. 8.4.3, p. 363) $\square$

Remark:
Let LR-DFA$^{allfinal}(\Gamma)$ be the automaton obtained from LR-DFA(Γ) by considering all states as final (accepting) states.
Lemma 2 says that LR-DFA$^{allfinal}(\Gamma)$ accepts exactly the viable prefixes of Γ.

Proof of LR(0) characterization

- **Left-to-Right-Direction:**
  LR(0) property implies that LR-DFA has no conflicts.

  Let q be a state of LR-DFA(Γ) with two items $[A \to \alpha.]$ and $[B \to \beta.\gamma]$.
  We show that these items do not cause a conflict.

  By Lemma 1(b), there are μ, ν with $\mu\alpha = \nu\beta$ and with $\mu = \varepsilon$ or $\nu = \varepsilon$.
  Let ψ be a path leading to q, then according to Lemma 1(a), there exists φ with $\psi = \varphi\mu\alpha = \varphi\nu\beta$.
  By Lemma 2, there are the following rightmost derivations

$$S \underset{rm}{\Longrightarrow}^{*} \varphi\mu A u \underset{rm}{\Longrightarrow} \varphi\mu\alpha u$$

$$S \underset{rm}{\Longrightarrow}^{*} \varphi\nu B v \underset{rm}{\Longrightarrow} \varphi\nu\beta\gamma v$$

Proof of LR(0) characterization (2)

Case 1: Suppose, the items $[A \to \alpha.]$ and $[B \to \beta.\gamma]$ are different and cause a reduce/reduce-conflict, i.e. $\gamma = \varepsilon$.

We show that then it has to hold that $A = B$ and $\alpha = \beta$, i.e. both items are identical which is a contradiction to the assumption.

Since $\gamma = \varepsilon$ and $\varphi\mu\alpha = \varphi\nu\beta$, it holds that $\varphi\mu\alpha = \varphi\nu\beta\gamma$.

By the LR(0) property, it holds $\varphi\mu = \varphi\nu$ and $A = B$ (and $v = v$).

From $\varphi\nu = \varphi\mu$ and $\varphi\mu\alpha = \varphi\nu\beta$, it follows that $\alpha = \beta$, i.e. both items are identical.

Proof of LR(0) characterization (3)

Case 2: Suppose the items $[A \to \alpha.]$ and $[B \to \beta.\gamma]$ cause a shift/reduce-conflict, i.e. $\gamma \neq \varepsilon$ and γ starts with a terminal symbol.

We consider the cases $\gamma \in T^+$ and $\gamma = cD\rho$, where in the first case γ contains no non-terminal, while in the second it contains at least one non-terminal.

If $\gamma \in T^+$, for $w = \gamma$, we obtain the derivation

$$S \underset{rm}{\Longrightarrow}^* \varphi\nu B v \underset{rm}{\Longrightarrow} \varphi\nu\beta w v = \varphi\mu\alpha w v$$

The LR(0) property yields $\varphi\nu = \varphi\mu$, $A = B$ and $v = wv$.
Thus, it has to hold that $w = \varepsilon$ which contradicts the assumption $\gamma \neq \varepsilon$.

Proof of LR(0) characterization (4)

If $\gamma = cD\rho$, we can extend the above rightmost derivation

$$\begin{array}{rll} S & \underset{rm}{\Longrightarrow}^{*} & \varphi\nu Bv \underset{rm}{\Longrightarrow} \varphi\nu\beta\gamma v = \varphi\mu\alpha cD\rho v \\ & \underset{rm}{\Longrightarrow}^{*} & \varphi\mu\alpha cxEyv \underset{rm}{\Longrightarrow} \varphi\mu\alpha cxwyv \end{array}$$

Then we have the rightmost derivations

$$S \underset{rm}{\Longrightarrow}^{*} \varphi\mu Au \underset{rm}{\Longrightarrow} \varphi\mu\alpha u$$

$$S \underset{rm}{\Longrightarrow}^{*} \varphi\mu\alpha cxEyv \underset{rm}{\Longrightarrow} \varphi\mu\alpha cxwyv$$

The LR(0) property yields in particular that $\varphi\mu\alpha cx = \varphi\mu$.

Thus, it has to hold that $\alpha cx = \varepsilon$ which is not possible; thus, the assumption yields a contradiction.

Proof of LR(0) characterization (5)

- **Right-to-Left Direction:** Conflict-freedom implies LR(0) property.

  According to the LR(0) definition, we consider two rightmost derivations

$$S \underset{rm}{\Longrightarrow}^* \psi A u \underset{rm}{\Longrightarrow} \psi\alpha u$$

$$S \underset{rm}{\Longrightarrow}^* \varphi B x \underset{rm}{\Longrightarrow} \psi\alpha y$$

  In derivation 2, we assume a production $B \to \delta$ such that $\varphi\delta x = \psi\alpha y$.

  By Lemma 2, $[A \to \alpha.]$ belongs to a state reached by $\psi\alpha$ and $[B \to \delta.]$ to a state reached by $\varphi\delta$.

  Since $\varphi\delta x = \psi\alpha y$, it holds that $\varphi\delta$ is prefix of $\psi\alpha$ or vice versa.

  Case distinction on the relationship between $\varphi\delta$ und $\psi\alpha$:

Proof of LR(0) characterization (6)

1. $\varphi\delta = \psi\alpha$:

   $[A \to \alpha.]$ and $[B \to \delta.]$ belong to the same state.
   Since the LR-DFA has no conflicts, it holds that $\alpha = \delta$ and $A = B$, thus also $\varphi = \psi$ and $x = y$. This yields the LR(0) property.

2. $\varphi\delta$ is a proper prefix of $\psi\alpha$:

   Since $\varphi\delta x = \psi\alpha y$, there exists $c \in T$ und $z \in T^*$ such that $x = czy$ and hence $\varphi\delta cz = \psi\alpha$.

   By Lemma 1(c), the state reached by $\varphi\delta$ has to contain a transition marked with c and an item $[C \to \mu.c\nu]$.

   Furthermore, by Lemma 2, the state reached by $\varphi\delta$ has to contain $[B \to \delta.]$ But this would cause a shift-reduce conflict which contradicts the conflict-freeness of LR-DFA such that this case cannot occur.

3. $\psi\alpha$ is a proper prefix of $\varphi\delta$: similar to case 2. $\square$

Example: Application of LR(0) characterization

Show (using the above theorem) that Γ_5 is LR(0).

Γ_5:

- $S \rightarrow A \mid B$
- $A \rightarrow aAb \mid 0$
- $B \rightarrow aBbb \mid 1$

Expressiveness of LR(k)

- For each context-free language L with the prefix property (i.e. $\forall v, w \in L$: v is no prefix of w), there exists an LR(0) grammar.
- Grammar Γ_5 is not LL(k), but LR(0).
- Methods for LR(1) can be generalized to LR(k), SLR(k) and LALR(k).

Resolving conflicts by lookahead

- Compute lookahead sets from $(N \cup T)^{\leq k}$ for items. The lookahead set of an item approximates the set of prefixes of length k with which the input rest at this item can start.
- If the lookahead sets at an item are disjoint, then the action to be executed (shift, reduce) can be determined by k symbols lookahead.
- For an item, select the action whose lookahead set contains the prefix of the input rest. Action table has to be extended.
- For computation of lookahead sets, there are different methods.

Common methods for lookahead computation

- SLR(k) uses LR-DFA and $FOLLOW_k$ of conflicting items for lookahead
- LALR(k) - lookahead LR - uses LR-DFA with state-dependent lookahead sets
- LR(k) integrates computation of lookahead sets in automata construction (LR(k) automaton)

SLR grammars

Definition (SLR(1) grammar)

Let $\Gamma = (N, T, \Pi, S)$ be a CFG and $LA([A \rightarrow \alpha.]) = FOLLOW_1(A)$.

A state LR-DFA(Γ) has an SLR(1) conflict if there exists

- two different reduce items with $LA([A \rightarrow \alpha.]) \cap LA([B \rightarrow \beta.]) \neq \emptyset$ or
- two items $[A \rightarrow \alpha.]$ and $[B \rightarrow \alpha.a\beta]$ with $a \in LA([A \rightarrow \alpha.])$.

Γ is SLR(1) if there is no SLR(1) conflict.

SLR grammars (2)

Example: Γ_6 is an SLR(1) grammar

- $S \rightarrow E\#$
- $E \rightarrow T + E \mid T$
- $T \rightarrow ID \mid N()$
- $N \rightarrow ID$

Consider the conflicts between $[E \rightarrow T.]$ and $[E \rightarrow T. + E]$ and between $[T \rightarrow ID.]$ and $[N \rightarrow ID.]$

$FOLLOW_1(E) \cap \{+\} = \{\#\} \cap \{+\} = \emptyset$
$FOLLOW_1(T) \cap FOLLOW_1(N) = \{\#, +\} \cap \{(\} = \emptyset$

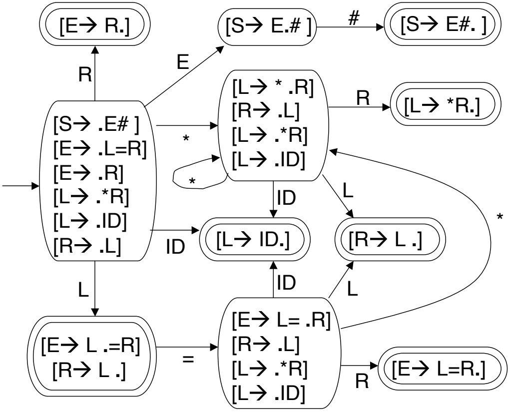
SLR grammars (3)

Example: Γ_7 (simplified C expressions) is **not** an SLR(1) grammar

- $S \rightarrow E\#$
- $E \rightarrow L = R \mid R$
- $L \rightarrow *R \mid ID$
- $R \rightarrow L$

SLR grammars (4)

LR-DFA for Γ_7



Only conflict in items $[E \to L. = R]$ and $[R \to L.]$ with

$$FOLLOW_1(R) \cap \{=\} = \{=, \#\} \cap \{=\} \neq \emptyset$$

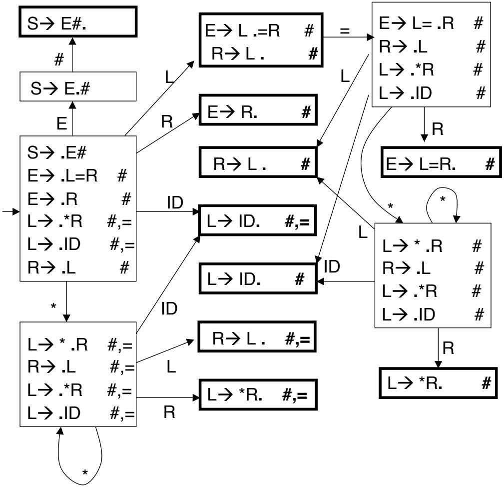
Construction of LR(1) automata

LR(1) automaton contains items $[A \rightarrow \alpha.\beta, V]$ with $V \subseteq T$ where

- α is on top of the stack
- the input rest is derivable from βc with $c \in V$, i.e. $V \subseteq FOLLOW_1(A)$.

Construction of LR(1) automata (2)

LR(1) automaton for Γ_7. Conflict is resolved, as $\{=\} \cap \{\#\} = \emptyset$.



LALR(1) automata

Informal description of LALR(1) automata:

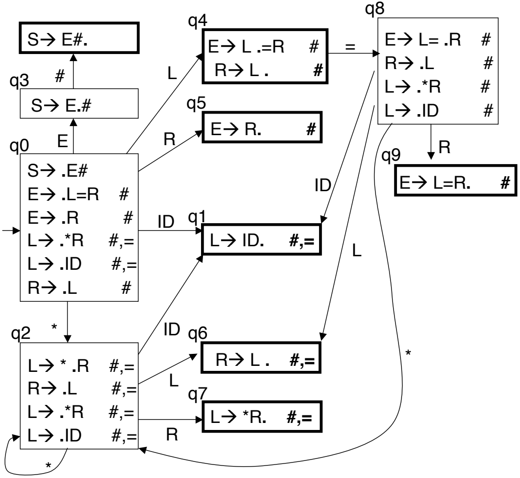
- LALR(1) automata can be constructed from LR(1) automata by merging states in which items only differ in lookahead sets.
- In the merged state, the items has the union of the lookahead sets as lookahead set.

Remarks:

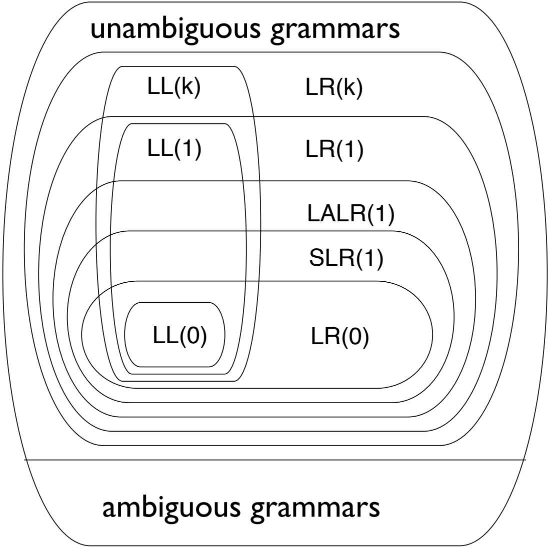
1. The LALR(1) automaton has the same states as the LR-DFA.
2. The LALR(1) automata can be constructed more efficiently.

Example: LALR(1) automaton

LALR(1) automaton for Γ_7.



Grammar classes



Literature

Recommended reading for bottom-up analysis:

- Wilhelm, Maurer: Chapter 8, Sections 8.4.1 - 8.4.5, pp. 353 - 383

2.2.3 Error Handling

Error Handling

Learning objectives:

- Problems and principles of error handling
- Techniques of error handling for context-free analysis

Principles of error handling

Error handling is required in all analysis phases and at runtime.

Types of errors:

- lexical errors
- parse errors (in context-free analysis)
- errors in name and type analysis
- runtime errors (cannot be avoided in most cases)
- logical errors (behavioral errors)

Remarks:

1. The first 2 (3) kinds of errors are syntactical errors. In the following, we only consider error handling in context-free analysis.
2. The specification of what an error is, is defined by the language specification.

Requirements for error handling

- Errors should be localized as precisely as possible.
  Problem: Errors are often not detected at the position where they were caused.
- As many errors as possible should be detected at once/in one phase/in one run.
  Problem: Avoid to report consecutive errors
- Errors are not always unique, i.e., it is not clear in general how to correct an error: `class int { Int a; .... }` or `int a = 1-;`
- Error handling should not slow down the analysis of correct programs.

Therefore, error handling is nontrivial and depends on the source language to be analysed.

Error handling in context-free analysis

1. **Panic error handling**
   Mark synchronizing terminal symbols, e.g. “`end`” or “;”

   If the parser reaches an error state, all symbols up to the next synchronizing symbol are skipped and the stack is corrected as if the production with the synchronizing symbol was read correctly.

   - Pros: easy to implement, termination guaranteed
   - Cons: large parts of the program can be skipped or misinterpreted
   - Example: Incorrect input `a : = b *** c;`
     Read until “;” correct stack and continue as if statement has been accepted

Error handling in context-free analysis (2)

2. **Error productions**
   Extend the grammar with productions describing typical error situations, so called error productions.
   Error messages can be directly associated with error productions.

   - Pros: easy to implement, termination guaranteed
   - Cons: extended grammar can belong to a more general grammar class; knowledge of typical error situations is necessary
   - Example: Typical error in PASCAL

     `if ...  then A := E; else ...`

     Error Production:

     `Stmt` $\rightarrow$ `if Expr then Stmt`* `; else Stmt`*

Error handling in context-free analysis (3)

3. **Production-local error correction**
   The goal is the local correction of the <u>input</u> such that the analysis can be resumed.
   Local means that an attempt is made to correct the input for the current production.

   - Pros: flexible and powerful technique
   - Cons: problematic if errors occur earlier than they can be detected; operations for corrections can lead to a nonterminating analysis

Error handling in context-free analysis (4)

4. **Global error correction**
   Attempt to get a correction that is as good as possible by altering the read input or the lookahead input.

   Idea: Define some distance or quality measure on inputs. For each incorrect input, look for a syntactically correct input that is best according to the used measure.

   - Pros: very powerful technique
   - Cons: analysis effort can be rather high; implementation is complex and poses risk of nontermination

Error handling in context-free analysis (5)

5. **Interactive error correction**
   In modern programming languages, syntactic analysis is often already supported by editors. In this case, editor marks error positions.
   - Pros: quick feedback; possible error positions are shown directly; interaction with the programmer is possible
   - Cons: editing can be disturbed; analysis must be able to handle incomplete programs

The presented techniques can be combined.
The selection criteria depend on the actual syntax.
Error handling also depends on the grammar class and implementation techniques used for the parser.

Burke-Fisher error handling

Example of a global error correction technique

- Procedure: Use a correction window of n symbols before the symbol at which the error was detected. Check all possible variations of symbol sequence in the correction window that can be obtained by insertion, deletion or replacement of a symbol at any position in the window.
- Quality measure: Choose the variation that allows the longest continuation of the parsing procedure.
- Implementation: Work with two stack automata, one representing the configuration at the beginning of the correction window, the other one the configuration at the end of the correction window. In an error case, the automaton running behind can be used to resume at the old position and to test the computed variations.

Literature

Recommended reading:
Wilhelm, Maurer: Chapter 8,
Sections 8.3.6 and 8.4.6 (general understanding sufficient)

M. G. Burke and G. a. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. ACM Transactions on Programming Languages and Systems, 9(2):164–197, Mar. 1987.

2.2.4 Concrete and Abstract Syntax

Concrete and Abstract Syntax

Learning objectives

- Connection of parsing to other phases of language processing and translation
- Differences between abstract and concrete syntax
- Language concepts for describing syntax trees
- Syntax tree construction

Connection of parsing to other phases

Several options:

1. Parsing directly controls the subsequent phases
2. Concrete syntax tree as interface
3. Abstract syntax tree as interface

Direct control by parser

- Parser calls other actions after each derivation/reduction step (Example: recursive descent parsing)
- Pros:
  - simple (if realizable)
  - flexible
  - efficient (especially memory efficient)
- Cons:
  - non-modular, no clear interfaces
  - not suitable for global aspects of translation
  - subsequent phases depend on parsing
  - cannot be used with every parser generator

Abstract syntax vs. Concrete syntax

Let *PL* be some (programming) language with CFG Γ and $p \in PL$ be a program.

Definition (Concrete syntax)

The concrete syntax of *PL* determines the actual text representation of programs (incl. key words, separators, etc.).
The syntax tree of p according to Γ is the concrete syntax tree of p.

Definition (Abstract syntax)

The abstract syntax of *PL* describes the tree structure of programs in a form that is sufficient and suitable for further processing.
A tree for representing a program p according to the abstract syntax is called the abstract syntax tree of p.

Abstract syntax

- Abstraction from keywords and separators
- Operator precedences are represented in the tree structure (different nonterminals are not necessary)
- Better incorporation of symbol information
- Simplifying transformations from concrete to abstract syntax might be applied

Remarks:

- The abstract syntax of a language is often not specified in the language report.
- The abstract syntax usually also comprises information about source code positions.

Example: Concrete vs. abstract syntax

Concrete syntax: Γ_2

- $S \rightarrow E\#$
- $E \rightarrow T + E \mid T$
- $T \rightarrow F * T \mid F$
- $F \rightarrow (E) \mid ID$

Abstract syntax

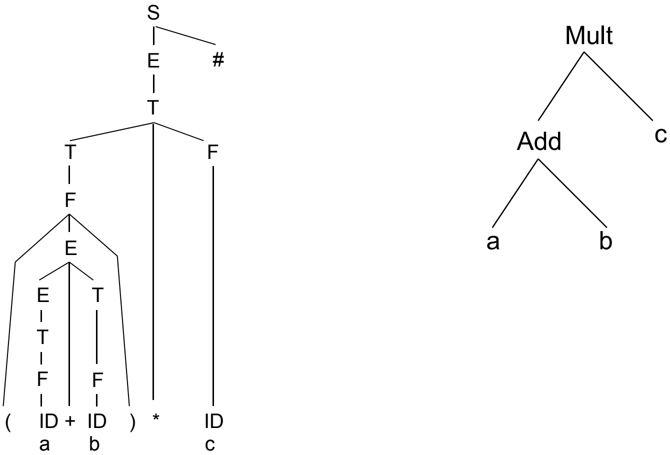
- Exp = Add | Mult | Ident
- Add (Exp left, Exp right)
- Mult (Exp left, Exp right)

Example: Concrete vs. abstract syntax (2)

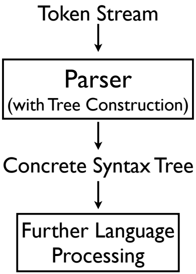
Text: $(a + b) * c$

Concrete syntax tree

Abstract syntax tree

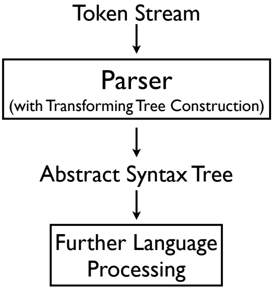


Concrete syntax tree as interface



- Resolves disadvantages of direct control by parser
- Advantages over abstract syntax
  - No additional specification of abstract syntax required
  - Tree construction does not have to be described
  - Tree construction can be done automatically by parser generators

Abstract syntax tree as interface



- Advantages over concrete syntax
  - Simpler, more compact tree representation
  - Simplifies later phases
  - Often implemented by programming or specification language as mutable data structure

Abstract syntax: Specification and tree construction

- For representing abstract syntax trees, we use order-sorted terms.
- The sets and types of these terms are described by type declarations.

Order-sorted data types

Definition

Order-sorted data types are specified by declarations of the following form:

- Variant type declarations $V = V_0 \mid V_1 \mid \ldots \mid V_m$
- Tuple type declaration $T(T_1 sel_1, \ldots, T_n sel_n)$
- List type declarations $L * S$

Example:

- Exp = Add | Mult | Ident
- Add (Exp left, Exp right)
  Mult (Exp left, Exp right)
- ExpList * Exp

where Ident is a predefined type.

Order-sorted data types (2)

Definition (Order-sorted types - contd.)

Order-sorted terms are recursively defined as

- If t is a term of type V_i, then it is also of type V.
- If t_i is a term of type T_i for each i, then $T(t_1, \ldots, t_n)$ is of type T, T is also the constructor.
- If $s_1, \ldots, s_n$ are terms of type S, then $L(s_1, \ldots, s_n)$ is of type L, L is also the list constructor.

Additional operators

- the selectors $sel_k : T \rightarrow T_k$ returns the k-th subterm
- the usual list operations (rest, append, conc, ...)

Order-sorted data types (3)

Remarks: Order-sorted data types

- generalize data types of functional languages by subtyping, a term can belong to several types
- are used in specification languages, e.g. OBJ3, Katja, ...
- are a very compact form for type declaration
- have a canonical implementation in OO languages

Example: Order-sorted data types and OO types

Declaration of order-sorted data types:

- Exp = Add | Mult | Neg | Ident
- Add (Exp left, Exp right)
- Mult (Exp left, Exp right)
- Neg (Exp val)

Implementation in Java

```
interface Exp {
  Exp left() throws IllegalSelectException;
  Exp right() throws IllegalSelectException;
  Exp val() throws IllegalSelectException;
}

class Add implements Exp {
  private Exp left;
  private Exp right;

  Add( Exp l, Exp r ) {
  left = l; right = r; }

  Exp left() { return left; }
  Exp right(){ return right; }
  Exp val() throws IllegalSelectException {
    throw new IllegalSelectException(); }
}
```

Implementation in Java (2)

```
class Mult implements Exp {
  // analog zu Add }

class Neg implements Exp {
  private Exp val;

  Neg( Exp v ) { val = v; }

  Exp left() throws IllegalSelectException {
    throw new IllegalSelectException(); }

  Exp right() throws IllegalSelectException {
    throw new IllegalSelectException(); }

  Exp val() { return val; }
}
```

Implementation in Java (3)

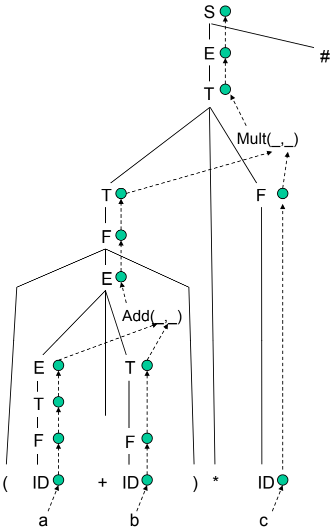
```
class Ident implements Exp extends PredefIdent {

  Exp left() throws IllegalSelectException {
    throw new IllegalSelectException(); }

  Exp right() throws IllegalSelectException {
    throw new IllegalSelectException();}

  Exp val() throws IllegalSelectException {
    throw new IllegalSelectException(); }
}
```

Transformation of concrete to abstract syntax



Specification of abstract syntax in JastAdd

```
Program ::= Exp;

abstract Exp;

abstract Binop:Exp ::= Left:Exp Right:Exp;
Plus:Binop;
Times:Binop;

Identifier:Exp ::= <Name>;
```

Transformation to abstract syntax with Beaver

```
%typeof S = "Program";
%typeof E = "Exp";
%typeof T = "Exp";
%typeof F = "Exp";

%goal S;
S = E.e                 {: return new Program(e); :};
E = T.le PLUS E.re      {: return new Plus(le,re); :}
  | T.t                 {: return t; :}
  ;
T = F.le TIMES T.re     {: return new Times(le,re); :}
  | F.f                 {: return f; :}
  ;
F = LPAREN E.e RPAREN   {: return e; :}
  | IDENTIFIER.id       {: return id; :}
  ;
```

Recommended reading

- Wilhelm, Maurer: Section 9.1, pp. 406 + 407
- Appel: Chapter 4, pp. 89 – 105