Languages and Compilers (SProg og Oversættere)

Bent Thomsen
Department of Computer Science
Aalborg University

Action Routines and Attribute Grammars

- Automatic tools can construct lexer and parser for a given context-free grammar
  - *E.g. JavaCC and JLex/CUP (and Lex/Yacc*)
- CFGs cannot describe all of the syntax of programming languages
  - An ad hoc techniques is to annotate the grammar with executable rules
  - These rules are known as *action routines*
- Action routines can be formalized **Attribute Grammars**
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammars

- Example: expressions of the form id + id
  - id's can be either int_type or real_type
  - types of the two id's must be the same
  - type of the expression must match it's expected type
- BNF:

  <expr> → <var> + <var>

  <var> → id
- Attributes:
  - actual_type - synthesized for **<var>** and **<expr>**
  - expected_type - inherited for **<expr>**

The Attribute Grammar

- Syntax rule: **<expr> → <var>[1] + <var>[2]**

  Semantic rules:

  <expr>.actual_type ← <var>[1].actual_type

  Predicate:

  <var>[1].actual_type == <var>[2].actual_type

  <expr>.expected_type == <expr>.actual_type

- Syntax rule: **<var> → id**

  Semantic rule:

  <var>.actual_type ← lookup (<var>.string)

Attribute Grammars

<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type

Attribute Grammars

- Def: An attribute grammar is a CFG G = (S, N, T, P) with the following additions:
  - For each grammar symbol x there is a set A(x) of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define synthesized attributes
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $i <= j <= n$, define inherited attributes
- Initially, there are intrinsic attributes on the leaves

Attribute Grammars

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.
  - Top-down grammars (LL(k)) generally require inherited flows

Attribute Grammars and Practice

- The attribute grammar formalism is important
  - Succinctly makes many points clear
  - Sets the stage for actual, *ad-hoc* practice
- The problems with attribute grammars motivate practice
  - Non-local computation
  - Need for centralized information (globals)
    - Advantages
      - Addresses the shortcomings of the AG paradigm
      - Efficient, flexible
    - Disadvantages
      - Must write the code with little assistance
      - Programmer deals directly with the details

The Realist's Alternative

Ad-hoc syntax-directed translation

- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- Allowing arbitrary code provides complete flexibility
  - Includes ability to do tasteless & bad things

To make this work

- Need names for attributes of each symbol on *lhs* & *rhs*
  - Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, …)
  - Yacc/CUP introduces **$$, $1, $2, … $n**, left to right
- Need an evaluation scheme
  - Fits nicely into **LR(1)** parsing algorithm

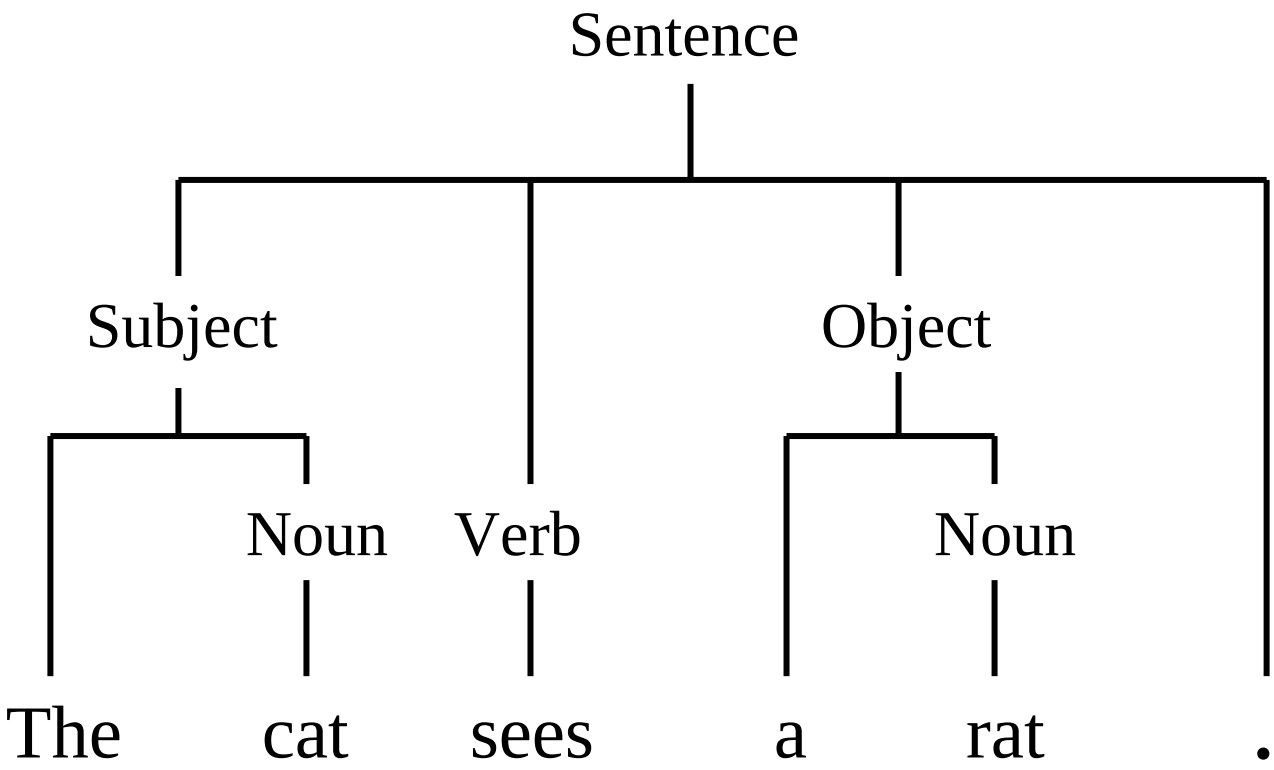
Back to Parsers – let us look at LR parsing

Generation of parsers

- We have seen that recursive decent parsers can be constructed automatically, e.g. JavaCC
- However, recursive decent parsers only work for LL(k) grammars
- Sometimes we need a more powerful language
- The LR languages are more powerful
- Parsers for LR languages a use bottom-up parsing strategy

Bottom up parsing

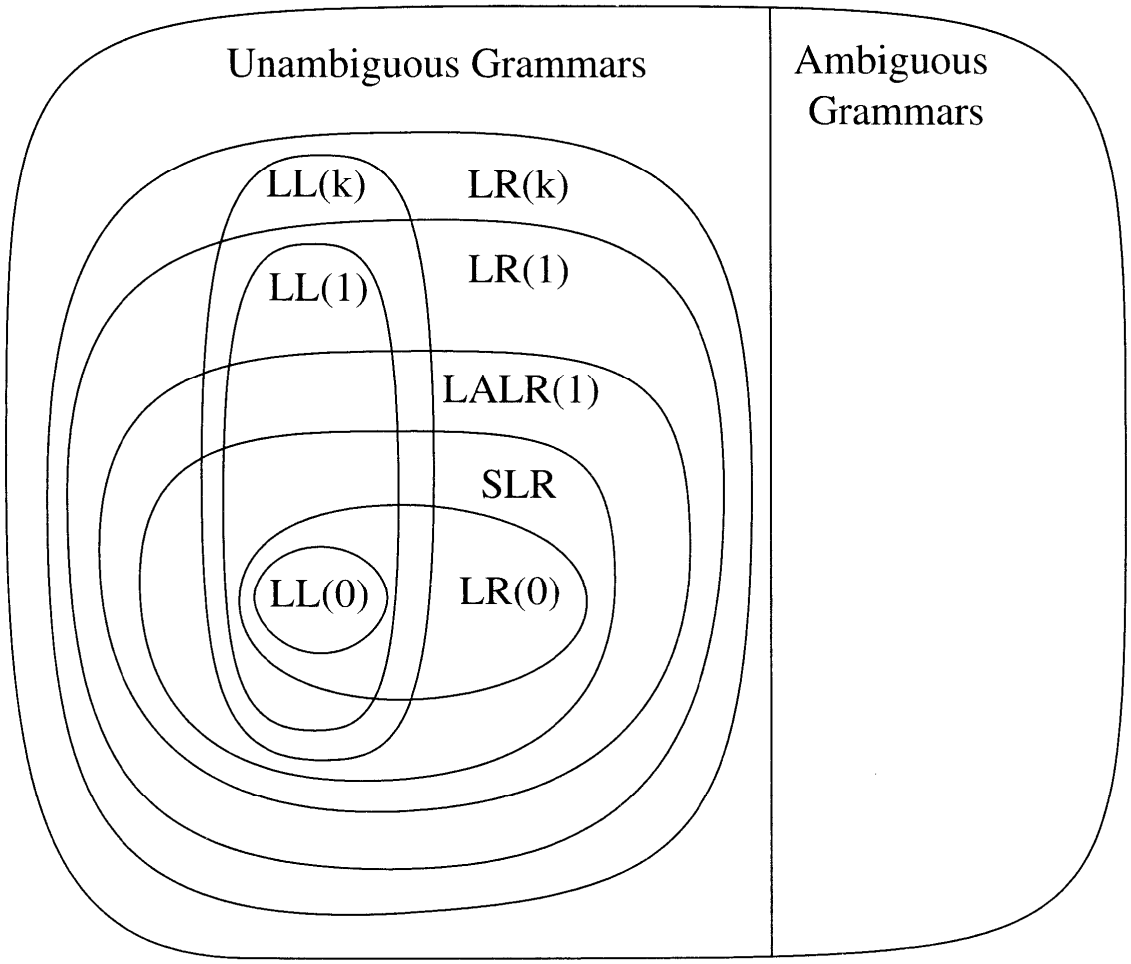
The parse tree “grows” from the bottom (leafs) up to the top (root).



Bottom Up Parsers

- Harder to implement than LL parser
  - but tools exist (e.g. JavaCUP and SableCC)
- Can recognize LR0, LR1, SLR, LALR grammars (bigger class of grammars than LL)
  - Can handle left recursion!
  - Usually more convenient because less need to rewrite the grammar.
- LR parsing methods are the most commonly used for automatic tools today (LALR in particular)

Hierarchy


Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)

Bottom Up Parsers: Overview of Algorithms

- LR0 : The simplest algorithm, theoretically important but rather weak (not practical)
- SLR : An improved version of LR0 more practical but still rather weak.
- LR1 : LR0 algorithm with extra lookahead token.
  - very powerful algorithm. Not often used because of large memory requirements (very big parsing tables)
- LALR : “Watered down” version of LR1
  - still very powerful, but has much smaller parsing tables
  - most commonly used algorithm today

Fundamental idea

- Read through every construction and recognize the construction in the end

- LR:
  - Left – the string is read from left to right
  - Right – we get a right derivation

- The parse tree is build from bottom up

Right derivations

```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```

Sentence

→ Subject Verb Object .

→ Subject Verb a Noun .

→ Subject Verb a rat .

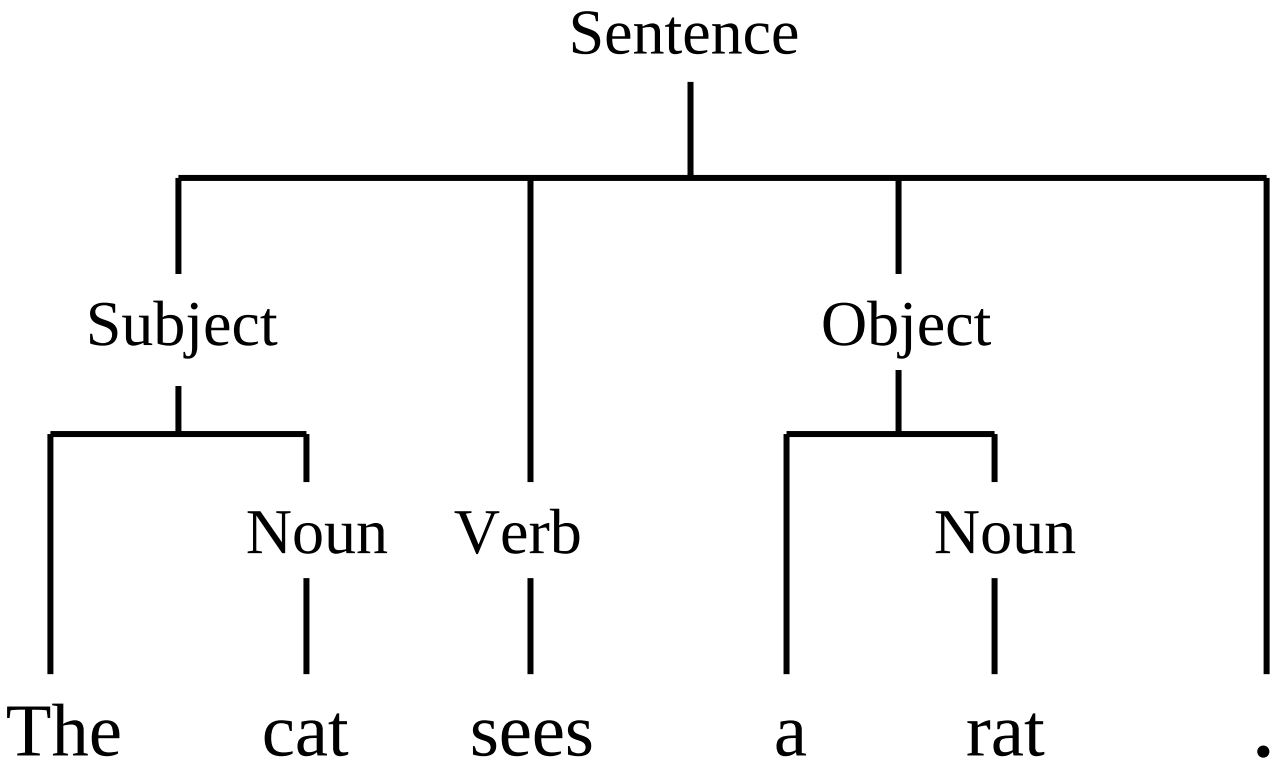
→ Subject sees a rat .

→ The Noun sees a rat .

→ The cat sees a rat .

Bottom up parsing

The parse tree “grows” from the bottom (leafs) up to the top (root).
Just read the right derivations backwards



Bottom Up Parsers

- All bottom up parsers have similar algorithm:
  - A loop with these parts:
    - try to find the leftmost node of the parse tree which has not yet been constructed, but all of whose children *have* been constructed. (This sequence of children is called a **handle**)
    - construct a new parse tree node. This is called **reducing**
- The difference between different algorithms is only in the way they find a handle.

Bottom-up Parsing

- Intuition about handles:
  - Def: β is the handle of the right sentential form

    $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$
  - Def: β is a phrase of the right sentential form

    γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$
  - Def: β is a simple phrase of the right sentential form

    γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$

Bottom-up Parsing

- Intuition about handles:
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning

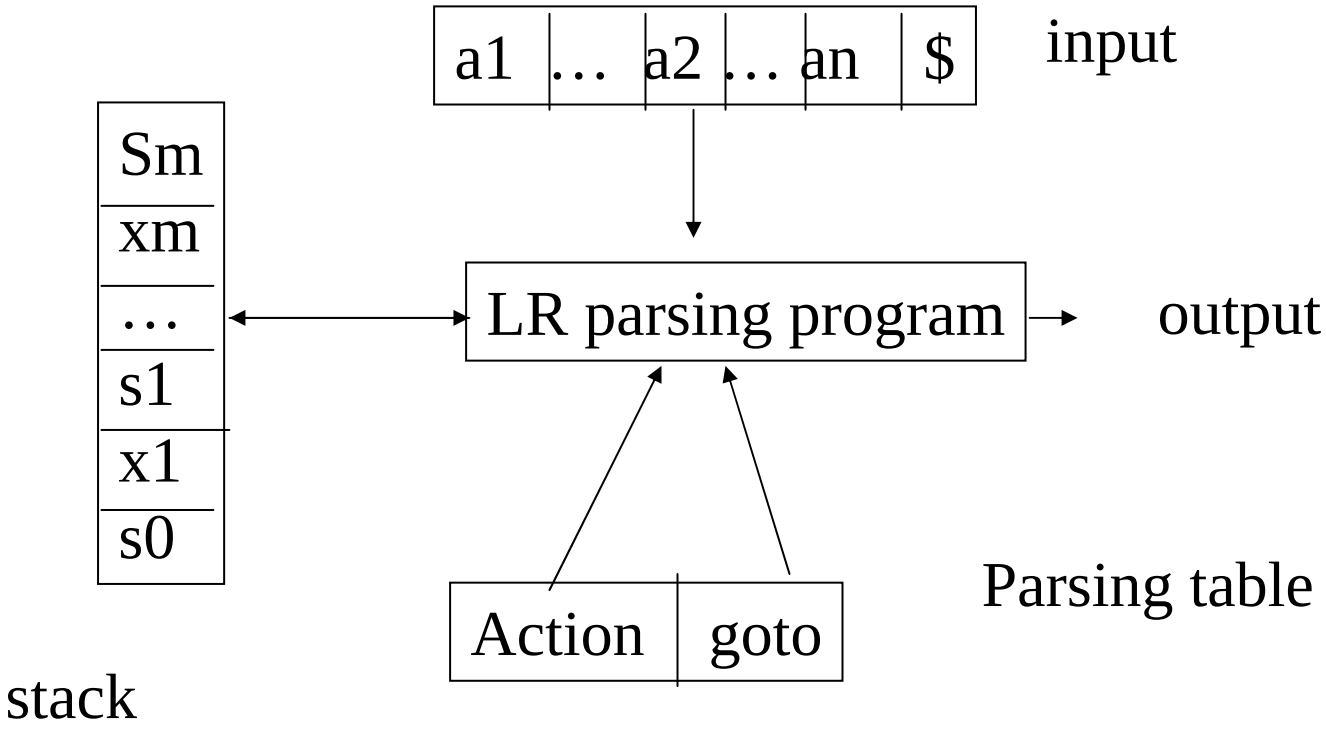
Bottom-up Parsing

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

The LR-parse algorithm

- A finite automaton
  - With transitions and states
- A stack
  - with objects (symbol, state)
- A parse table

Model of an LR parser:



si is a state, xi is a grammar symbol

All LR parsers use the same algorithm, different grammars have different parsing table.

The parse table

- For every state and every terminal
  - either shift x

    Put next input-symbol on the stack and go to state x
  - or reduce production

    On the stack we now have symbols to go backwards in the production – afterwards do a goto
- For every state and every non-terminal
  - Goto x

    Tells us, in which state to be in after a reduce-operation
- Empty cells in the table indicates an error

Example-grammar

- (0) S’ $\rightarrow$ S$
  - This production *augments* the grammar
- (1) S $\rightarrow$ (S)S
- (2) S $\rightarrow$ ε

- This grammar generates all expressions of matching parentheses

Example - parse table

	(	)	$	*S'*	*S*
0	s2	r2	r2		g1
1		s3	r0		
2	s2	r2	r2		g3
3		s4			
4	s2	r2	r2		g5
5		r1	r1		

By reduce we indicate the number of the production
r0 = accept
Never a goto by *S′*

Example – parsing

Stack	Input	Action
$\$_0$	()()$	shift 2
$\$_0(_2$	)()$	reduce S→ε
$\$_0(_2S_3$	)()$	shift 4
$\$_0(_2S_3)_4$	()$	shift 2
$\$_0(_2S_3)_4(_2$	)$	reduce S→ε
$\$_0(_2S_3)_4(_2S_3$	)$	shift 4
$\$_0(_2S_3)_4(_2S_3)_4$	$	reduce S→ε
$\$_0(_2S_3)_4(_2S_3)_4S_5$	$	reduce S→(S)S
$\$_0(_2S_3)_4S_5$	$	reduce S→(S)S
$\$_0S_1$	$	reduce S'→S

The resultat

- Read the productions backwards and we get a right derivation:
- S’ $\Rightarrow$ S $\Rightarrow$ (S)S $\Rightarrow$(S)(S)S
  $\Rightarrow$(S)(S) $\Rightarrow$ (S)() $\Rightarrow$()()

LR(0)-items

Item : A production with a selected position (marked by a point)

X $\rightarrow\alpha.\beta$ indicates that at the stack we have α and the first of the input can be derived from β

Our example grammar has the following items:

S' $\rightarrow$.S$	S' $\rightarrow$S.$	(S' $\rightarrow$S$.)
S $\rightarrow$.(S)S	S$\rightarrow$(.S)S	S$\rightarrow$(S.)S
S$\rightarrow$(S).S	S$\rightarrow$(S)S.	S$\rightarrow$.

LR(0)-DFA

- Every state is a set of items
- Transitions are labeled by symbols

- States must be *closed*

- New states are constructed from states and transitions

Closure(I)

Closure(I) =

repeat
for any item $A \rightarrow \alpha.X\beta$ in I
for any production $X \rightarrow \gamma$
$I \leftarrow I \cup \{ X \rightarrow .\gamma \}$
unto I does not change
return I

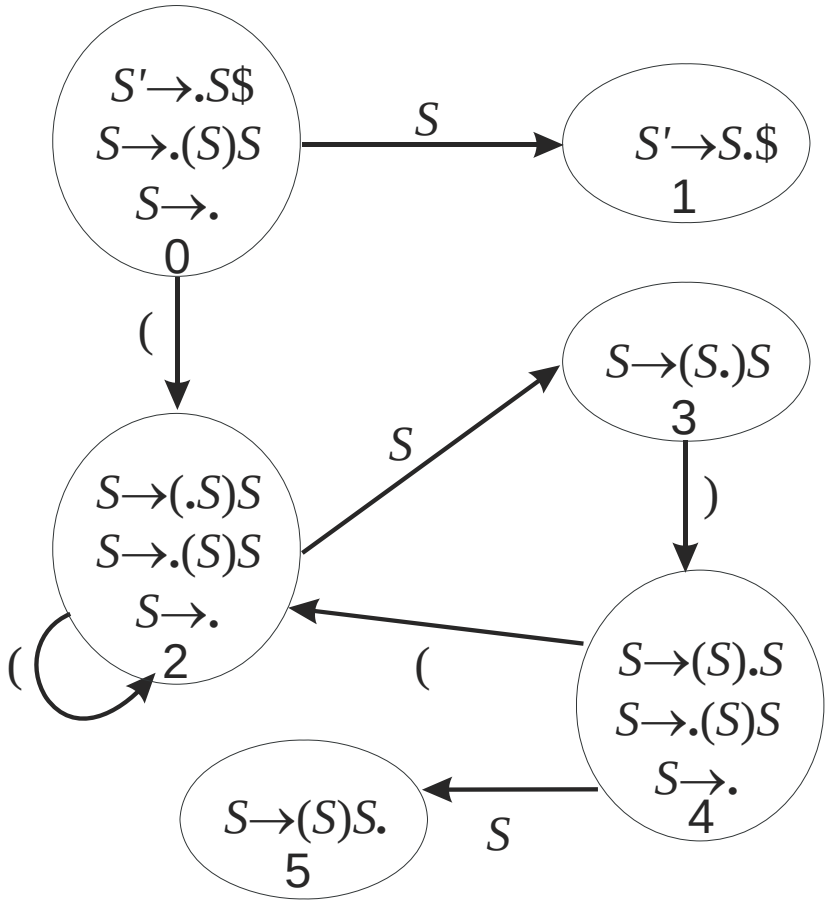
Goto(I,X)

Describes the X-transition from the state I

Goto(I,X) =

Set J to the empty set
for any item $A \rightarrow \alpha.X\beta$ in I
add $A \rightarrow \alpha X.\beta$ to J
return Closure(J)

The DFA for our grammar



LR(0)-parse table

- state I with x-transition (x terminal) to J
  - shift J in cell (I,x)
- state I with final item ($X \rightarrow \alpha.$) corresponding to the productionen n
  - reduce n in all celler (I,x) for all terminals x
- state I with X-transition (x non-terminal) to J
  - goto J in cell (I,X)
- empty cells - error

Shift-reduce-conflicts

- What happens, if there is a shift and a reduce in the same cell
  - so we have a shift-reduce-conflict
  - and the grammar is not LR(0)

- Our example grammar is not LR(0)

Shift-reduce-conflicts

	(	)	$	*S'*	*S*
0	**s2/r2**	r2	r2		g1
1	r0	**s3/r0**	r0		
2	**s2/r2**	r2	r2		g3
3		s4			
4	**s2/r2**	r2	r2		g5
5	r1	r1	r1		

LR0 Conflicts

The LR0 algorithm doesn't always work. Sometimes there are "problems" with the grammar causing LR0 conflicts.

An LR0 conflict is a situation (DFA state) in which there is more than one possible action for the algorithm.

More precisely there are two kinds of conflicts:

- shift <-> reduce
  - When the algorithm cannot decide between a shift action or a reduce action
- reduce <-> reduce
  - When the algorithm cannot decide between two (or more) reductions (for different grammar rules).

Parser Conflict Resolution

Most programming language grammars are LR 1. But, in practice, one still encounters grammars which have parsing conflicts.

=> a common cause is an **ambiguous grammar**

Ambiguous grammars always have parsing conflicts (because they are ambiguous this is just unavoidable).

In practice, parser generators still generate a parser for such grammars, using a “resolution rule” to resolve parsing conflicts deterministically.

=> The resolution rule may or may not do what you want/expect

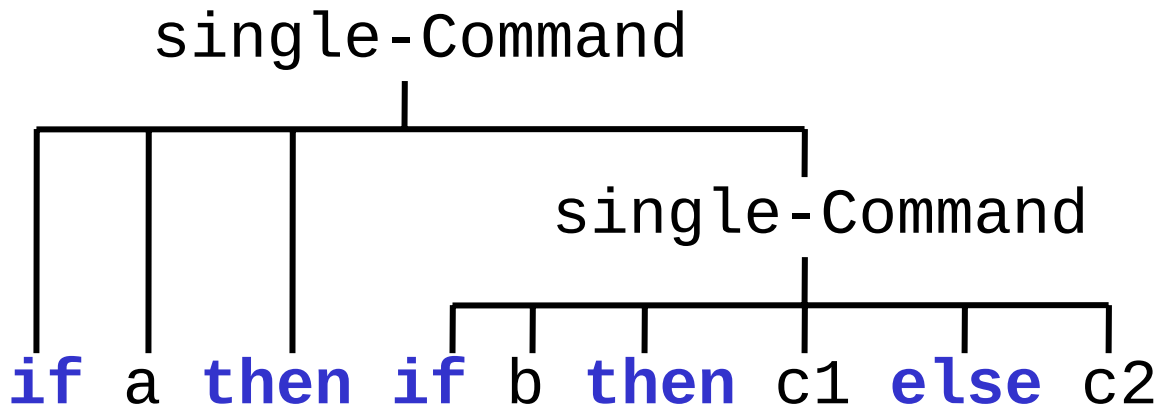
=> You will get a warning message. If you know what you are doing this can be ignored. Otherwise => try to solve the conflict by disambiguating the grammar.

Parser Conflict Resolution

Example: (from Mini-triangle grammar)

```
single-Command
  ::= if Expression then single-Command
    | if Expression then single-Command
                    else single-Command
```

This parse tree?

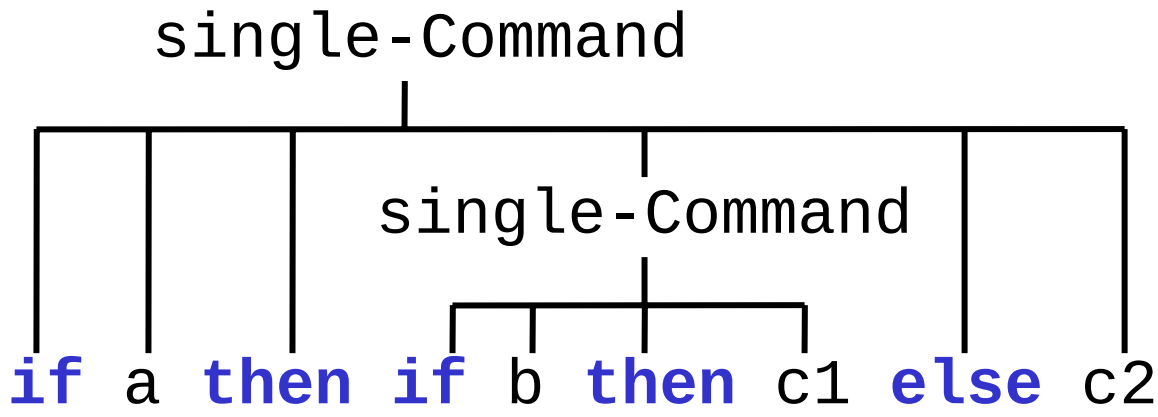


Parser Conflict Resolution

Example: (from Mini-triangle grammar)

```
single-Command
  ::= if Expression then single-Command
    | if Expression then single-Command
                    else single-Command
```

or this one ?



Parser Conflict Resolution

Example: “dangling-else” problem (from Mini-triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                     else single-Command
```

LR1 items (in some state of the parser)

```
sC   ::= if E then sC • {… else …}
sC   ::= if E then sC •  else sC {…}
```

Shift-reduce conflict!

Resolution rule: shift has priority over reduce.

Q: Does this resolution rule solve the conflict? What is its effect on the parse tree?

Parser Conflict Resolution

There is usually also a resolution rule for shift-reduce conflicts, for example the rule which appears first in the grammar description has priority.

Reduce-reduce conflicts usually mean there is a real problem with your grammar.

=> You need to fix it! Don’t rely on the resolution rule!

LR(0) vs. SLR

- LR(0) - here we do not look at the next symbol in the input before we decide whether to shift or to reduce

- SLR - here we do look at the next symbol
  - reduce $X \rightarrow \alpha$ is only necessary, when the next terminal y is in follow(X)
  - this rule removes at lot of potential s/r- og r/r-conflicts

SLR

- DFA as the LR(0)-DFA
- the parse table is a bit different:
  - shift and goto as with LR(0)
  - reduce $X \rightarrow \alpha$ only in cells (X,w) with $w \in \text{follow}(X)$
  - this means fewer reduce-actions and so fewer conflicts

LR(1)

- Items are now pairs ($A \rightarrow \alpha.\beta$, x)
  - x is an arbitrary terminal
  - means, that the top of the stack is α and the input can be derivered from βx
  - Closure-operation is different
  - Goto is (more or less) the same
  - The initial state is generated from (S' $\rightarrow$.S$, ?)

LR(1)-the parse table

- Shift and goto as before
- Reduce
  - state I with item ($A \rightarrow \alpha.$, z) gives a reduce $A \rightarrow \alpha$ in cell (I,z)

- LR(1)-parse tables are very big

Example

0: S' $\rightarrow$ S$

1: S $\rightarrow$ V=E

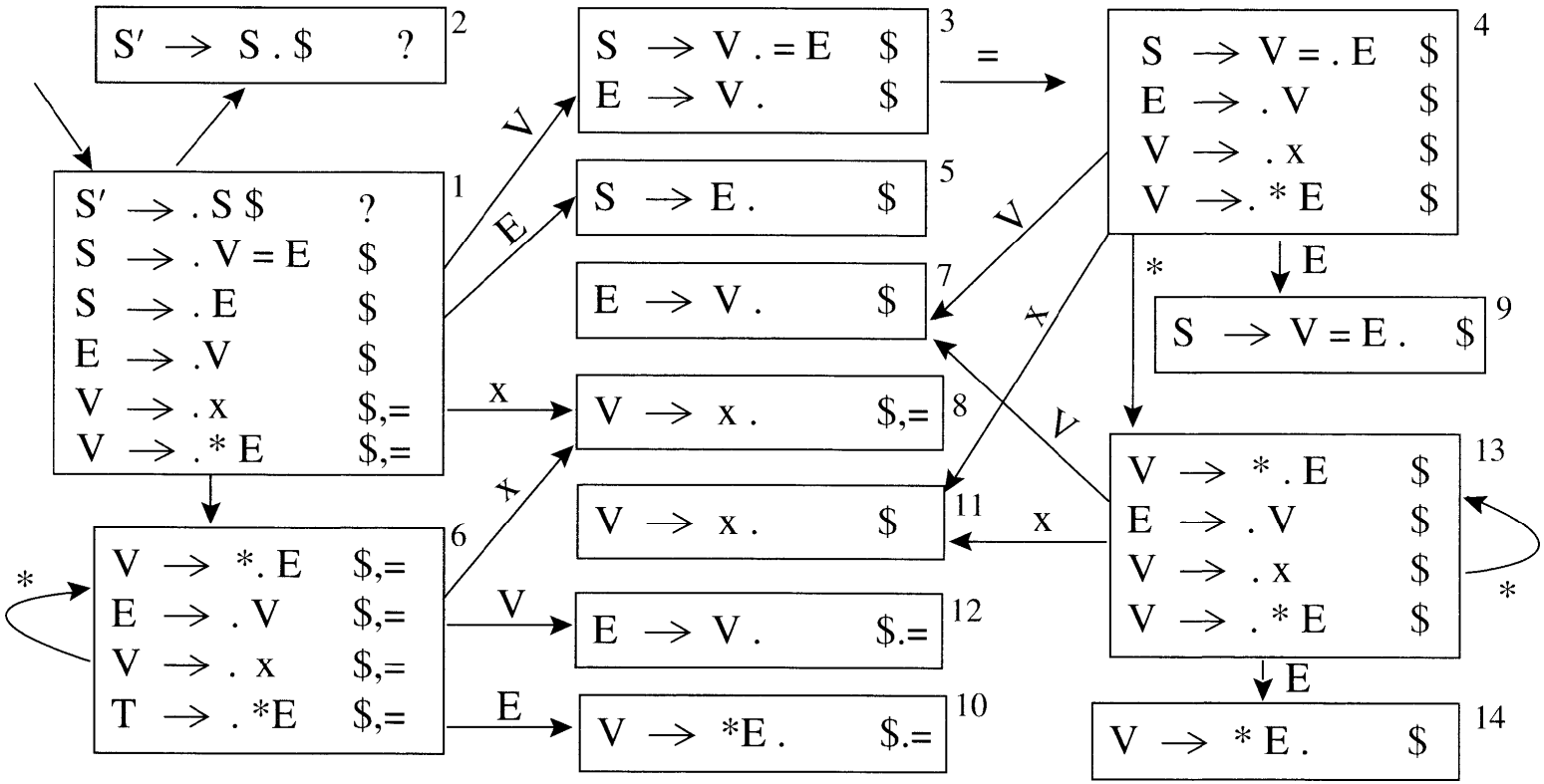
2: S $\rightarrow$ E

3: E $\rightarrow$ V

4: V $\rightarrow$ x

5: V $\rightarrow$ *E

LR(1)-DFA


S′ → S . $? 2
S′ → . S $?
S → . V = E $
S → . E $
E → . V $
V → . x $,=
V → . * E $,= 1
S → V . = E $
E → V . $ 3
S → E . $ 5
E → V . $ 7
V → x . $,= 8
V → x . $ 11
E → V . $.= 12
V → * E . $.= 10
V → * . E $,=
E → . V $,=
V → . x $,=
T → . * E $,= 6
S → V = . E $
E → . V $
V → . x $
V → . * E $ 4
S → V = E . $ 9
V → * . E $
E → . V $
V → . x $
V → . * E $ 13
V → * E . $ 14
V
E
x
=
*

LR(1)-parse table

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			

	x	*	=	$	S	E	V
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

LALR(1)

- A variant of LR(1) - gives smaller parse tables
- We allow ourselves in the DFA to combine states, where the items are the same except the *x*.
- In our example we combine the states
  - 6 and 13
  - 7 and 12
  - 8 and 11
  - 10 and 14

LALR(1)-parse-table

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s8	s6				g9	g7
5							
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

4 kinds of parsers

- 4 ways to generate the parse table
- LR(0)
  - Easy, but only a few grammars are LR(0)
- SLR
  - Relativey easy, a bit more grammars are SLR
- LR(1)
  - Difficult, but alle common languages are LR(1)
- LALR(1)
  - A bit difficult, but simpler and more efficient than LR(1)
  - In practice all grammars are LALR(1)

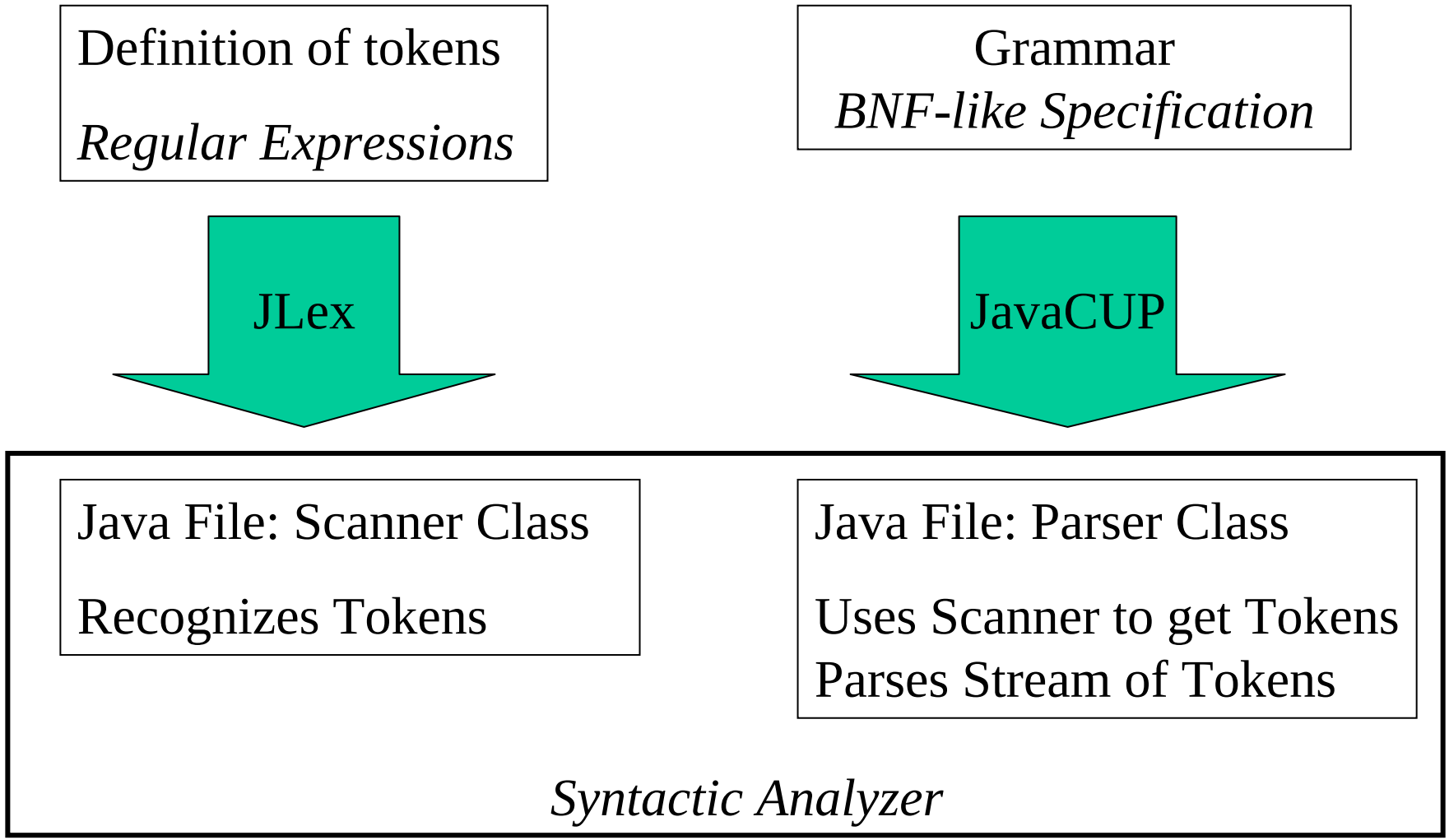
Enough background!

- All of this may sound a bit difficult (and it is)
- But it can all be automated!
- Now lets talk about tools
  - CUP (or Yacc for Java)
  - SableCC

Java Cup

- Accepts specification of a CFG and produces an LALR(1) parser (expressed in Java) with action routines expressed in Java
- Similar to yacc in its specification language, but with a few improvements (better name management)
- Usually used together with JLex

JavaCUP: A LALR generator for Java



Steps to use JavaCup

- Write a javaCup specification (cup file)
  - Defines the grammar and actions in a file (say, calc.cup)
- Run javaCup to generate a parser
  - java java_cup.Main < calc.cup
  - Notice the package prefix;
  - notice the input is standard in;
  - Will generate parser.java and sym.java (default class names, which can be changed)
- Write your program that uses the parser
  - For example, UseParser.java
- Compile and run your program

Java Cup Specification Structure

```
java_cup_spec ::= package_spec
                  import_list
                  code_part
                  init_code
                  scan_code
                  symbol_list
                  precedence_list
                  start_spec
                  production_list
```

- Great, but what does it mean?
  - Package and import controls Java naming
  - Code and init_code allows insertion of code in generated output
  - Scan code specifies how scanner (lexer) is invoked
  - Symbol list and precedence list specify terminal and non-terminal names and their precedence
  - Start and production specify grammar and its start point

Calculator javaCup specification (calc.cup)

```
terminal           PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr PLUS expr
        | expr MINUS expr
        | expr TIMES expr
        | expr DIVIDE expr
        | LPAREN expr RPAREN
        | NUMBER
      ;
```

- Is the grammar ambiguous?
- How can we get PLUS, NUMBER, ...?
  - They are the terminals returned by the scanner.
- How to connect with the scanner?

ambiguous grammar error

- If we enter the grammar

  Expression ::= Expression PLUS Expression;
- without precedence JavaCUP will tell us:

  Shift/Reduce conflict found in state #4

  between Expression ::= Expression PLUS Expression .

  and Expression ::= Expression . PLUS Expression

  under symbol PLUS

  Resolved in favor of shifting.
- The grammar is ambiguous!
- Telling JavaCUP that PLUS is left associative helps.

Corresponding scanner specification (calc.lex)

```
import java_cup.runtime.*;
%%
%implements java_cup.runtime.Scanner
%type Symbol
%function next_token
%class CalcScanner
%eofval{ return null;
%eofval}
NUMBER = [0-9]+
%%
"+" { return new Symbol(CalcSymbol.PLUS); }
"-" { return new Symbol(CalcSymbol.MINUS); }
"*" { return new Symbol(CalcSymbol.TIMES); }
"/" { return new Symbol(CalcSymbol.DIVIDE); }
{NUMBER} { return new Symbol(CalcSymbol.NUMBER, new Integer(yytext()));}
\r\n {}
. {}
```

- Connection with the parser
  - imports java_cup.runtime.*, Symbol, Scanner.
  - implements Scanner
  - next_token: defined in Scanner interface
  - CalcSymbol, PLUS, MINUS, ...
  - new Integer(yytext())

Run JLex

java JLex.Main calc.lex

- note the package prefix JLex
- program text generated: calc.lex.java

javac calc.lex.java

- classes generated: CalcScanner.class

Generated CalcScanner class

```
1. import java_cup.runtime.*;
2. class CalcScanner implements java_cup.runtime.Scanner {
3. ... ....
4.   public Symbol next_token () {
5.  ... ...
6.       case 3: { return new Symbol(CalcSymbol.MINUS); }
7.       case 6: { return new Symbol(CalcSymbol.NUMBER, new
   Integer(yytext()));}
8.       ... ...
9.    }
10. }
```

- Interface Scanner is defined in java_cup.runtime package

```
public interface Scanner {
    public Symbol next_token() throws java.lang.Exception;
}
```

Run javaCup

- Run javaCup to generate the parser
  - java java_cup.Main -parser CalcParser -symbols CalcSymbol < calc.cup
  - classes generated:
    - CalcParser;
    - CalcSymbol;
- Compile the parser and relevant classes
  - javac CalcParser.java CalcSymbol.java CalcParserUser.java
- Use the parser
  - java CalcParserUser

The token class Symbol.java

```
1.  public class Symbol {
2.     public int sym, left, right;
3.     public Object value;
4.     public Symbol(int id, int l, int r, Object o) {
5.        this(id); left = l; right = r;  value = o;
6.     }
7.     ... ...
8.     public Symbol(int id, Object o) {  this(id, -1, -1, o); }
9.     public String toString() { return "#"+sym; }
10. }
```

- Instance variables:
  - sym: the symbol type;
  - left: left position in the original input file;
  - right: right position in the original input file;
  - value: the lexical value.
- Recall the action in lex file:

```
return new Symbol(CalcSymbol.NUMBER, new Integer(yytext()));}
```

CalcSymbol.java (default name is sym.java)

```
1.  public class CalcSymbol {
2.    public static final int MINUS = 3;
3.    public static final int DIVIDE = 5;
4.    public static final int NUMBER = 8;
5.    public static final int EOF = 0;
6.    public static final int PLUS = 2;
7.    public static final int error = 1;
8.    public static final int RPAREN = 7;
9.    public static final int TIMES = 4;
10.   public static final int LPAREN = 6;
11. }
```

- Contain token declaration, one for each token (terminal); Generated from the terminal list in cup file
  - terminal PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
  - terminal Integer NUMBER
- Used by scanner to refer to symbol types (e.g., return new Symbol(CalcSymbol.PLUS);
- Class name comes from –symbols directive.
  - java java_cup.Main -parser CalcParser -symbols CalcSymbol calc.cup

The program that uses the CalcPaser

```
import java.io.*;
class CalcParserUser {
   public static void main(String[] args){
      try {
         File inputFile = new File ("calc.input");
         CalcParser parser= new CalcParser(new CalcScanner(new FileInputStream(inputFile)));
         parser.parse();
        } catch (Exception e) {  e.printStackTrace();
        }
    }
}
```

- The input text to be parsed can be any input stream (in this example it is a FileInputStream);
- The first step is to construct a parser object. A parser can be constructed using a scanner.
  - this is how scanner and parser get connected.
- If there is no error report, the expression in the input file is correct.

Evaluate the expression

- The previous specification only indicates the success or failure of a parser. No semantic action is associated with grammar rules.
- To calculate the expression, we must add java code in the grammar to carry out actions at various points.
- Form of the semantic action:

```
expr:e1 PLUS expr:e2
 {: RESULT = new Integer(e1.intValue()+ e2.intValue());    :}
```

  - Actions (java code) are enclosed within a pair {: :}
  - Labels e2, e2: the objects that represent the corresponding terminal or non-terminal;
  - RESULT: The type of RESULT should be the same as the type of the corresponding non-terminals. e.g., expr is of type Integer, so RESULT is of type integer.

Change the calc.cup

```
terminal            PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr:e1 PLUS expr:e2 {: RESULT = new Integer(e1.intValue()+ e2.intValue());    :}
        | expr:e1 MINUS expr:e2  {: RESULT = new Integer(e1.intValue()- e2.intValue());  :}
        | expr:e1 TIMES expr:e2  {: RESULT = new Integer(e1.intValue()* e2.intValue());  :}
        | expr:e1 DIVIDE expr:e2  {: RESULT = new Integer(e1.intValue()/ e2.intValue());  :}
        | LPAREN expr:e RPAREN {: RESULT = e;    :}
        | NUMBER:e {: RESULT= e; :}
```

Change CalcPaserUser

```
import java.io.*;
class CalcParserUser {
  public static void main(String[] args){
    try {
      File inputFile = new File ("calc.input");
      CalcParser parser= new CalcParser(new CalcScanner(new FileInputStream(inputFile)));
      Integer result= (Integer)parser.parse().value;
      System.out.println("result is "+ result);
     } catch (Exception e) {  e.printStackTrace();
     }
   }
}
```

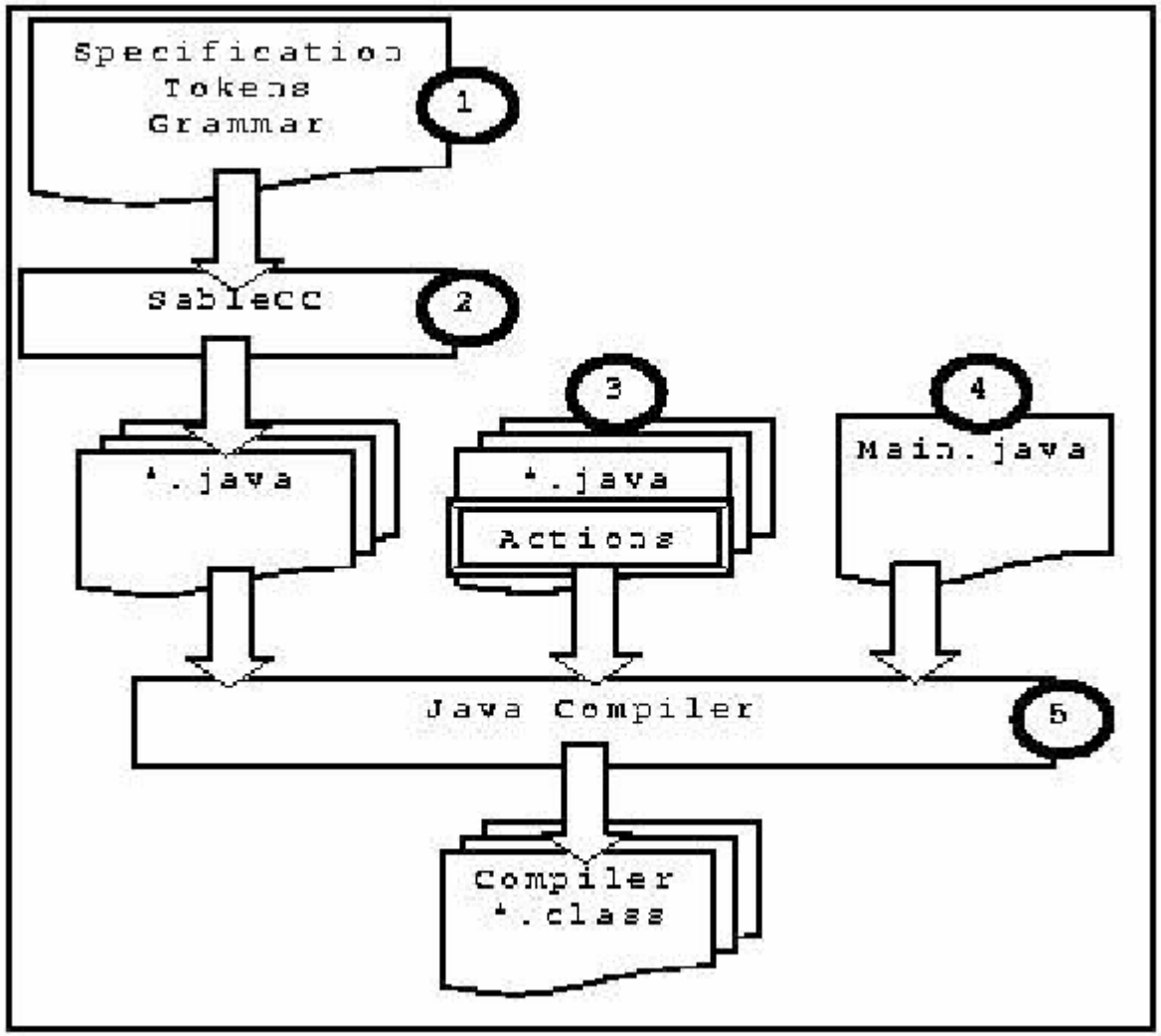
- Why the result of parser().value is an Integer?
  - This is determined by the type of expr, which is the head of the first production in javaCup specification:

    non terminal Integer expr;

SableCC

- Object Oriented compiler framework written in Java
  - There are also versions for C++ and C#
- Front-end compiler compiler like JavaCC and JLex + CUP
- Lexer generator based on DFA
- Parser generator based on LALR(1)
- Object oriented framework generator:
  - Strictly typed Abstract Syntax Tree
  - Tree-walker classes
  - Uses inheritance to implement actions
  - Provides visitors for user manipulation of AST
    - E.g. type checking and code generation

Steps to build a compiler with SableCC



1. Create a SableCC specification file
2. Call SableCC
3. Create one or more working classes, possibly inherited from classes generated by SableCC
4. Create a Main class activating lexer, parser and working classes
5. Compile with Javac

SableCC Example

```
Package Prog
Helpers
  digit = ['0' .. '9'];
  tab = 9;  cr = 13;  lf = 10;
  space = ' ';
  graphic = [[32 .. 127] + tab];

Tokens
  blank = (space | tab | cr | lf)* ;
  comment = '//' graphic* (cr | lf);
  while = 'while';
  begin = 'begin';
  end = 'end';
  do = 'do';
  if = 'if';
  then = 'then';
  else = 'else';
  semi = ';';
  assign = '=';
  int = digit digit*;
  id = ['a'..'z'](['a'..'z']|['0'..'9'])*;

Ignored Tokens
  blank, comment;
```

```
Productions
  prog = stmlist;

  stm = {assign} [left:]:id assign [right]:id |
        {while} while id do stm |
        {begin} begin stmlist end |
        {if_then} if id then stm;

stmlist = {stmt} stm |
          {stmtlist} stmlist semi stm;
```

SableCC output

- The *Lexer* package containing the Lexer and LexerException classes
- The *parser* package containing the Parser and ParserException classes
- The *node* package contains all the classes defining typed AST
- The *analysis* package containing one interface and three classes mainly used to define AST walkers based on the visitors pattern

Syntax Tree Classes for Prog

For each non-terminal in the grammar, SableCC generates an abstract class, for example:

```
abstract class PProg extends Node {}
```

where **`Node`** is a pre-defined class of syntax tree nodes which provides some general functionality.

Similarly we get abstract classes **`PStm`** and **`PStmlist`**.

The names of these classes are systematically generated from the names of the non-terminals.

Syntax Tree Classes for Prog

For each production, SableCC generates a class, for example:

```
class AAssignStm extends PStm
{
    PTerm _left_;
    PTerm _right_;

    public void apply(Switch sw)
    {
        ((Analysis) sw).caseAAssignStm(this);
    }
}
```

There are also **`set`** and **`get`** methods for **`_left_`** and **`_right_`**, constructors, and other housekeeping methods which we won't use.

Using SableCC's Visitor Pattern

The main way of using SableCC's visitor pattern is to define a class which extends **`DepthFirstAdapter`**.

By over-riding the methods **`inAAssignStm`** or **`outAAssignStm`** etc. we can specify code to be executed when entering or leaving each node during a depth first traversal of the syntax tree.

If we want to modify the order of traversal then we can over-ride **`caseAAssignStm`** etc. but this is often not necessary.

The **`in`** and **`out`** methods return **`void`**, but the class provides **`HashTable in, out;`** which we can use to store types of expressions.

A SableCC Grammar with transformations

SableCC specification of tokens:

```
Package expression;
Helpers
    digit = ['0' .. '9'];
    tab = 9;
    cr = 13;
    lf = 10;
    eol = cr lf | cr | lf; // This takes care of different platforms

    blank = (' ' | tab | eol)+;
Tokens
    l_par = '(';
    r_par = ')';
    plus = '+';
    minus = '-';
    mult = '*';
    div = '/';
    comma = ',';

    blank = blank;
    number = digit+;
Ignored Tokens
    blank;
```

A SableCC Grammar with transformations

Followed by the productions:

```
Productions

    grammar = exp_list {-> New grammar([exp_list.exp])};

    exp_list {-> exp*} = exp exp_list_tail* {-> [exp exp_list_tail.exp]};

    exp_list_tail {-> exp} = comma exp {-> exp};

    exp =
          {plus}   exp plus factor  {-> New exp.plus(exp, factor.exp)  }
        | {minus}  exp minus factor {-> New exp.minus(exp, factor.exp) }
        | {factor} factor           {-> factor.exp}
    ;

    factor {-> exp} =
          {mult} factor mult term {-> New exp.mult(factor.exp, term.exp )}
        | {div}  factor div term  {-> New exp.div(factor.exp, term.exp ) }
        | {term} term             {-> term.exp}
    ;

    term {-> exp} =
          {number} number          {-> New exp.number(number)}
        | {exp}    l_par exp r_par {-> exp}
    ;
```

A SableCC Grammar with transformations

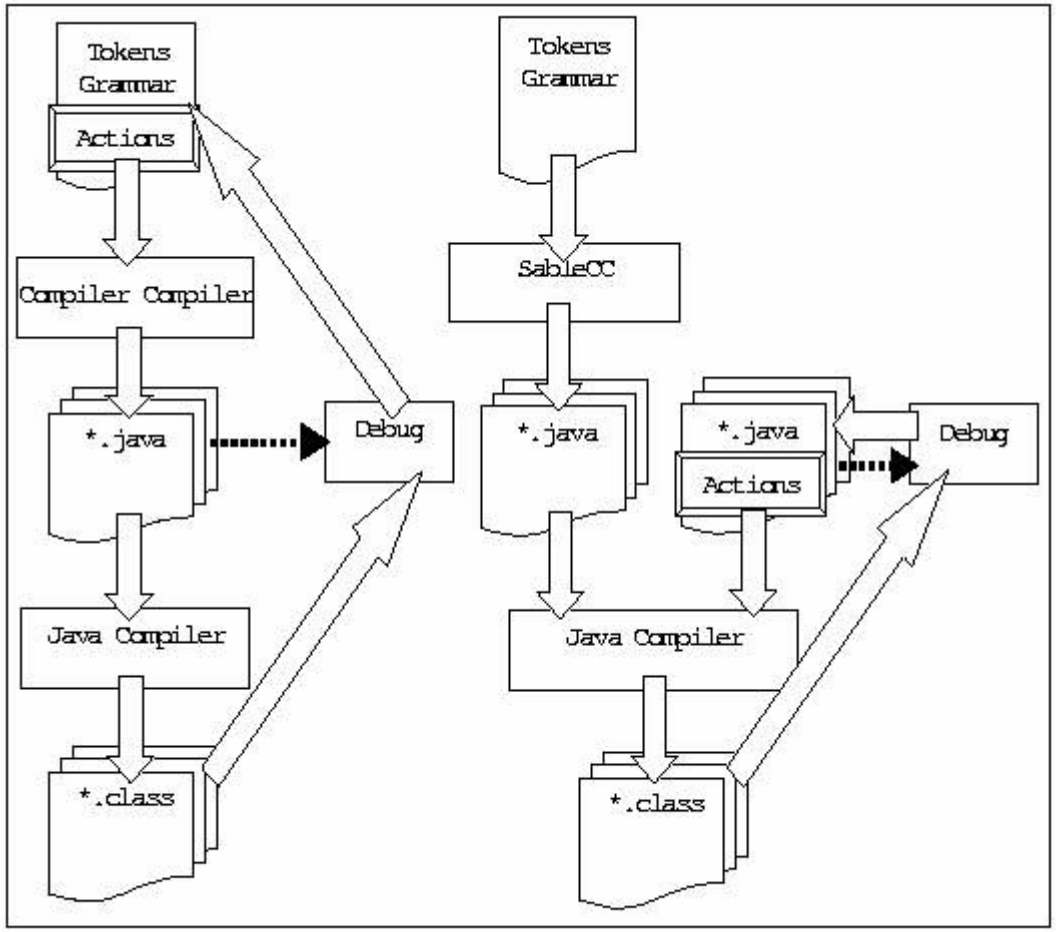
Followed by the Abstract Syntax Tree definition:

```
Abstract Syntax Tree

    grammar = exp* ;

    exp = {plus}   [l]:exp [r]:exp
    | {minus}  [l]:exp [r]:exp
    | {div}    [l]:exp [r]:exp
    | {mult}   [l]:exp [r]:exp
    | {number} number
    ;
```

JLex/CUP vs. SableCC



Advantages of SableCC

- Automatic AST builder for multi-pass compilers
- Compiler generator out of development cycle when grammar is stable
- Easier debugging
- Access to sub-node by name, not position
- Clear separation of user and machine generated code
- Automatic AST pretty-printer
- Version 3.0 Allow declarative grammar transformations

This completes our tour of the compiler front-end

What to do now?

- If your language is simple and you want to be in complete control, build recursive decent parser by hand
- If your language is LL(k) use JavaCC
- If your language is LALR(1) and most languages are!
  - Either use JLex/CUP (Lex/Yacc or SML-Lex/SML-Yacc)
  - Or use SableCC
  - Solve shift-reduce conflicts
- It is a really good idea to produce an AST
- Use visitors pattern on AST to do more work
  - Contextual analysis
  - Type checking
  - Code generation

Now let’s talk a bit about programming language design

..\..\..\Download\HeilsbergDeclarative.wmv

Programming Language Design

- The Art
  - The creative process
- The Science
  - Theoretical and experimental results showing what is possible
    - Tokens can be described by RE and implemented by DFA
    - LL Parsing can be implemented by Recursive Decent
    - LR Parsing can be implemented using a table driven Push Down automaton
- The Engineering
  - The putting it all together in a sensible way, I.e.
    - Choosing which parsing strategy to use (LL vs. LALR)
    - Implementing by hand or via tool
    - Choosing good data-structure

Syntax Design Criteria

- Readability
  - syntactic differences reflect semantic differences
  - verbose, redundant
- Writeability
  - concise
- Ease of verifiability
  - simple semantics
- Ease of translation
  - simple language
  - simple semantics
- Lack of ambiguity
  - dangling else
  - Fortran's A(I,J)

Lexical Elements

- Character set
- Identifiers
- Operators
- Keywords
- Noise words
- Elementary data
  - numbers
    - integers
    - floating point
  - strings
  - symbols
- Delimiters
- Comments
- Blank space
- Layout
  - Free- and fixed-field formats

Some nitty gritty decisions

- Primitive data
  - Integers, floating points, bit strings
  - Machine dependent or independent (standards like IEEE)
  - Boxed or unboxed
- Character set
  - ASCII, EBCDIC, UNICODE
- Identifiers
  - Length, special start symbol (#,$...), type encode in start letter
- Operator symbols
  - Infix, prefix, postfix, precedence
- Comments
  - REM, /* …*/, //, !, …
- Blanks
- Delimiters and brackets
- Reserved words or Keywords

Syntactic Elements

- Definitions
- Declarations
- Expressions
- Statements

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Overall Program Structure

- Subprograms
  - shallow definitions
    - C
  - nested definitions
    - Pascal
- Data (OO)
  - shallow definitions
    - C++, Java, Smalltalk
- Separate Interface
  - C, Fortran
  - ML, Ada
- Mixed data and programs
  - C
  - Basic
- Others
  - Cobol
    - Data description separated from executable statements
    - Data and procedure division

Keep in mind

There are many issues influencing the design of a new programming language:

- Choice of paradigm
- Syntactic preferences
- Even the compiler implementation
  - e.g no of passes
  - available tools

There are many issues influencing the design of new compiler:

- No of passes
- The source, target and implementation language
- Available tools

Some advice from an expert

- Programming languages are for people
- Design for yourself and your friends
- Give the programmer as much control as possible
- Aim for brevity