1. Overview

Short History of Compiler Construction

Formerly "a mystery", today one of the best-known areas of computing

1957	Fortran	first compilers (arithmetic expressions, statements, procedures)
1960	Algol	first formal language definition (grammars in Backus-Naur form, block structure, recursion, ...)
1970	Pascal	user-defined types, virtual machines (P-code)
1985	C++	object-orientation, exceptions, templates
1995	Java	just-in-time compilation

We only look at imperative languages

Functional languages (e.g. Lisp) and logical languages (e.g. Prolog) require different techniques.

Why should I learn about compilers?

It's part of the general background of a software engineer

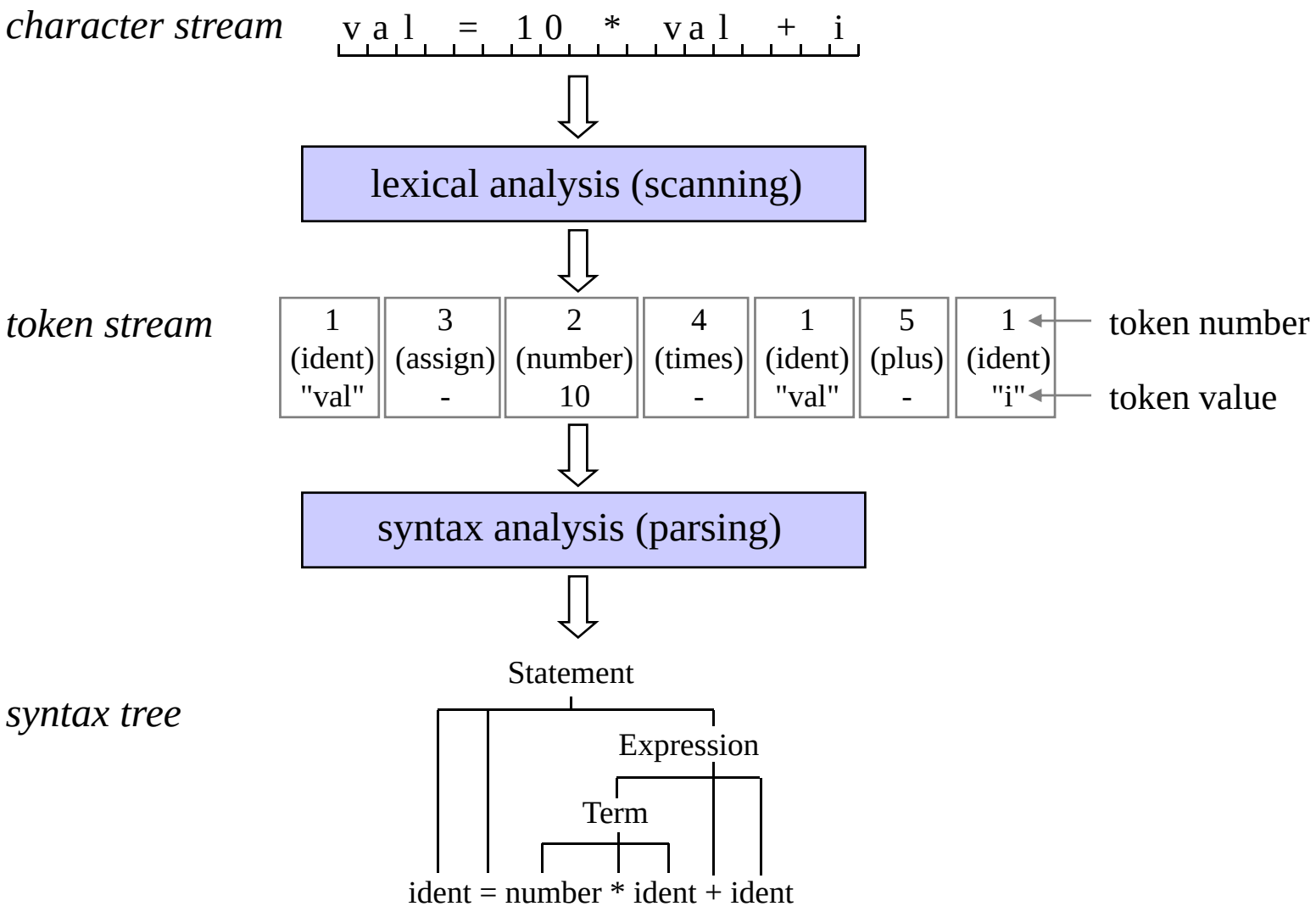
- How do compilers work?
- How do computers work?
  (instruction set, registers, addressing modes, run-time data structures, ...)
- What machine code is generated for certain language constructs?
  (efficiency considerations)
- What is good language design?
- Opportunity for a non-trivial programming project

Also useful for general software development

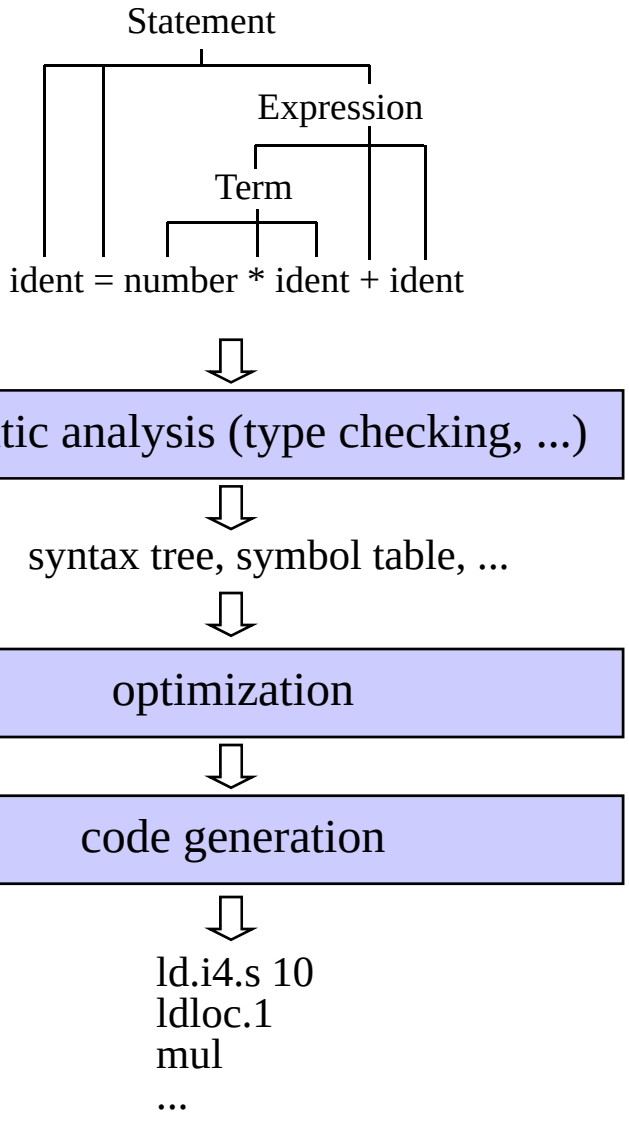
- Reading syntactically structured command-line arguments
- Reading structured data (e.g. XML files, part lists, image files, ...)
- Searching in hierarchical namespaces
- Interpretation of command codes
- ...

1. Overview

Dynamic Structure of a Compiler

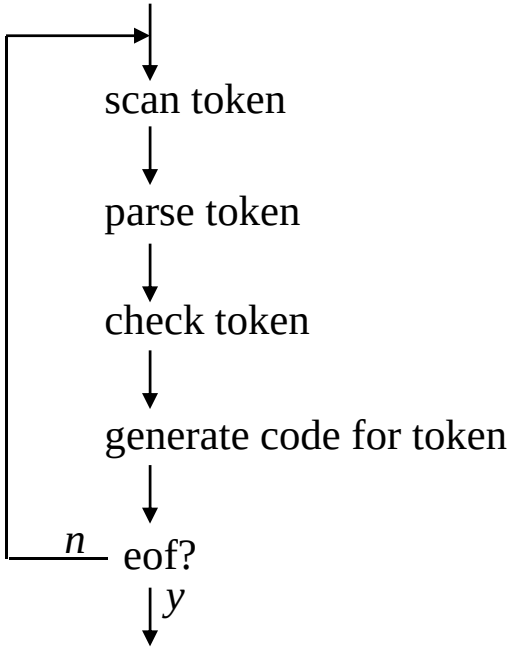


Dynamic Structure of a Compiler



Single-Pass Compilers

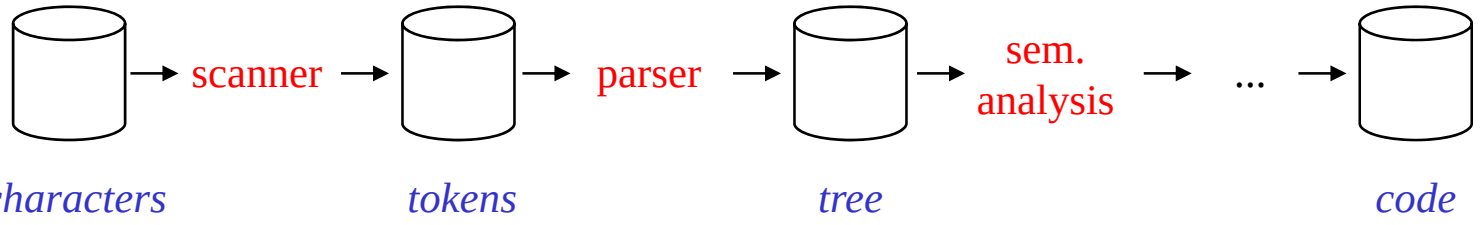
Phases work in an interleaved way



The target program is already generated while the source program is read.

Multi-Pass Compilers

Phases are separate "programs", which run sequentially

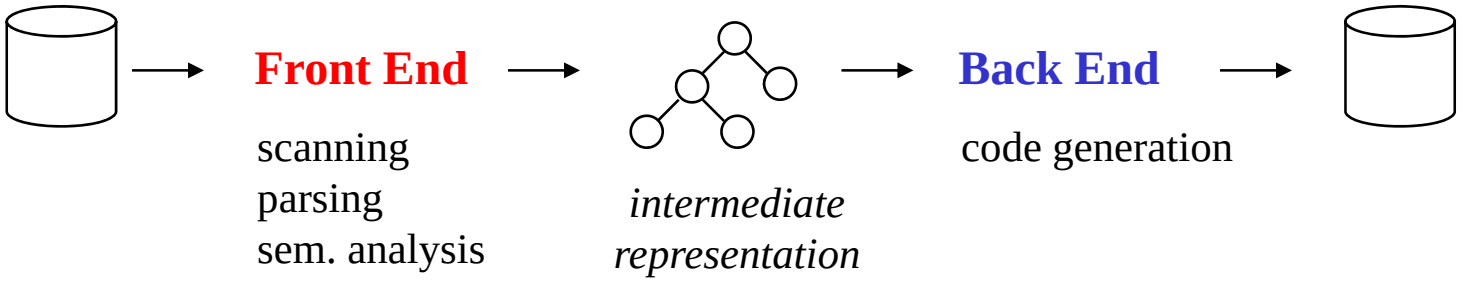


Each phase reads from a file and writes to a new file

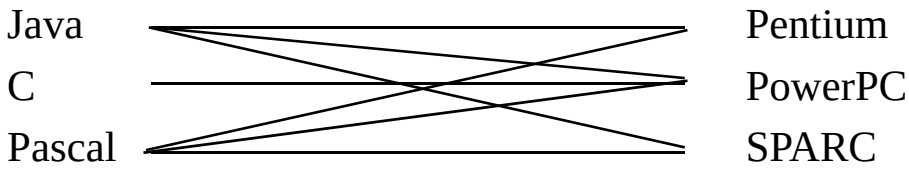
Why multi-pass?

- if memory is scarce (irrelevant today)
- if the language is complex
- if portability is important

Today: Often Two-Pass Compilers



language-dependent **machine-dependent**



any combination possible

Advantages

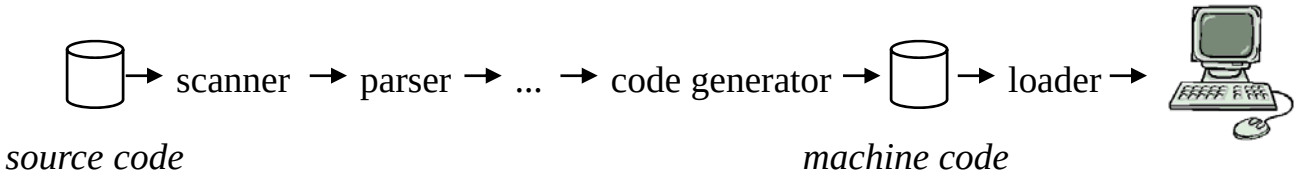
- better portability
- many combinations between front ends and back ends possible
- optimizations are easier on the intermediate representation than on source code

Disadvantages

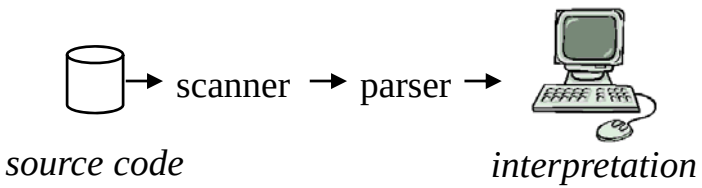
- slower
- needs more memory

Compiler versus Interpreter

Compiler translates to machine code

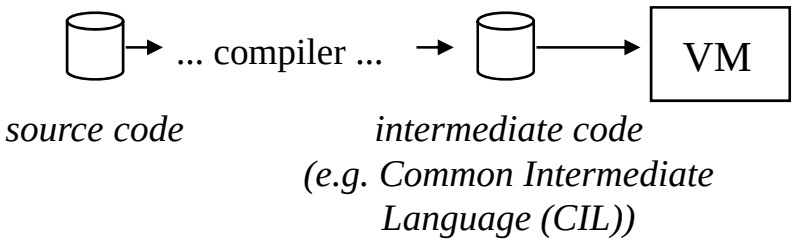


Interpreter executes source code "directly"



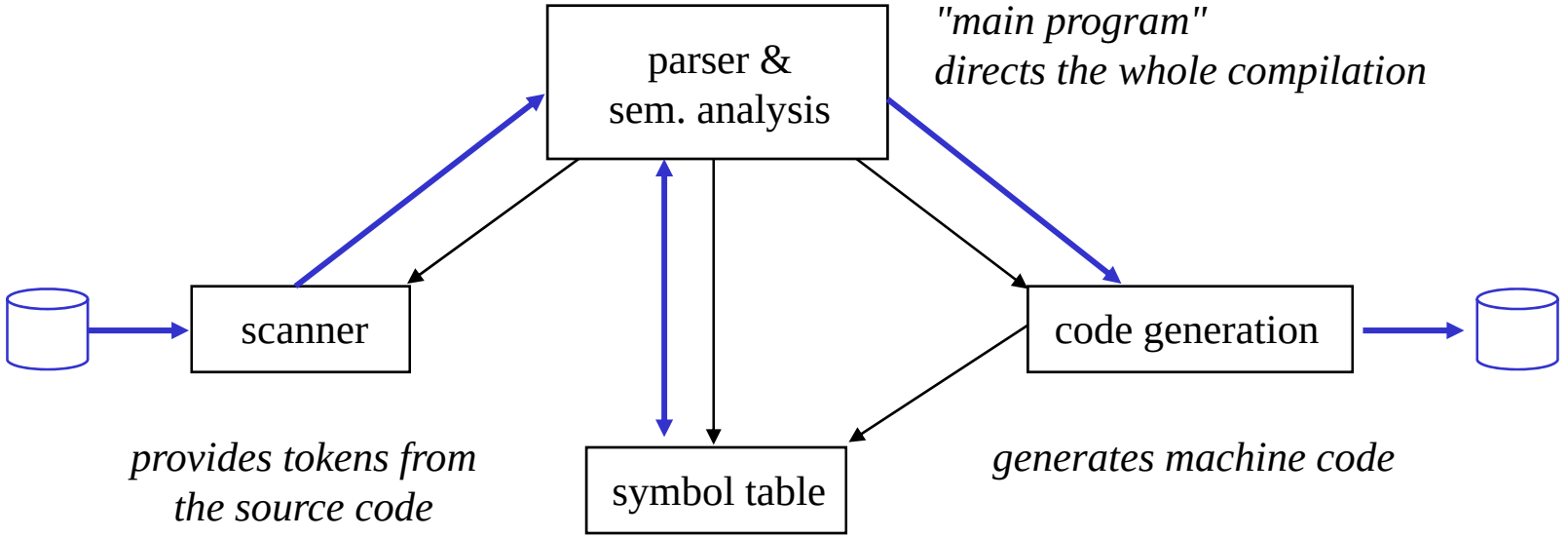
- statements in a loop are scanned and parsed again and again

Variant: interpretation of intermediate code



- source code is translated into the code of a *virtual machine* (VM)
- VM interprets the code simulating the physical machine

Static Structure of a Compiler



uses

data flow

1. Overview

What is a grammar?

Example

```
Statement = "if" "(" Condition ")" Statement ["else" Statement].
```

Four components

terminal symbols	are atomic	"if", ">=", ident, number, ...
nonterminal symbols	are derived into smaller units	Statement, Expr, Type, ...
productions	rules how to decompose nonterminals	Statement = Designator "=" Expr ";". Designator = ident ["." ident]. ...
start symbol	topmost nonterminal	CSharp

EBNF Notation

Extended Backus-Naur form

John Backus: developed the first Fortran compiler
Peter Naur: edited the Algol60 report

symbol	*meaning*	*examples*
string	denotes itself	"=", "while"
name	denotes a T or NT symbol	ident, Statement
=	separates the sides of a production	A = b c d .
.	terminates a production	
|	separates alternatives	a | b | c 🕓 a or b or c
(...)	groups alternatives	a (b | c) 🕓 ab | ac
[...]	optional part	[a] b 🕓 ab | b
{...}	repetitive part	{ a } b 🕓 b | ab | aab | aaab | ...

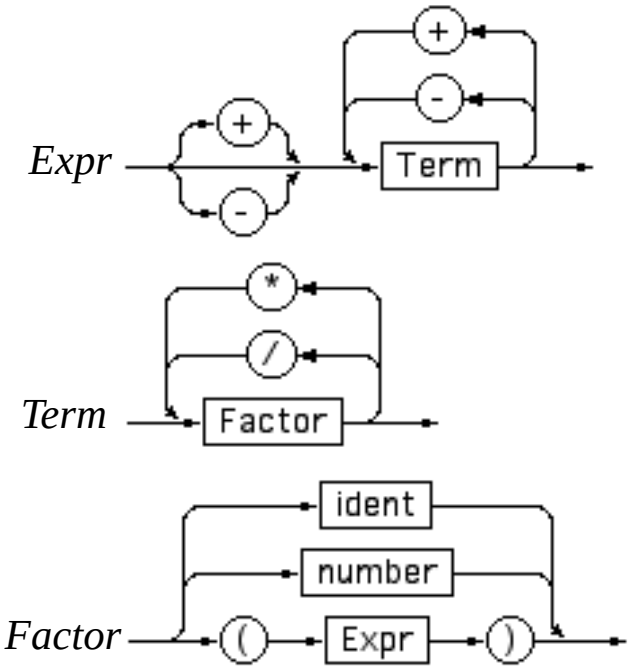
Conventions

- terminal symbols start with lower-case letters (e.g. ident)
- nonterminal symbols start with upper-case letters (e.g. Statement)

Example: Grammar for Arithmetic Expressions

Productions

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```



Terminal symbols

simple TS: "+", "-", "*", "/", "(", ")" (just 1 instance)

terminal classes: ident, number (multiple instances)

Nonterminal symbols

Expr, Term, Factor

Start symbol

Expr

Operator Priority

Grammars can be used to define the priority of operators

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```

input: - a * 3 + b / 4 - c

- - ident * number + ident / number - ident
- - Factor * Factor + Factor / Factor - Factor
- - Term + Term - Term
- Expr

"*" and "/" have higher priority than "+" and "-"

"-" does not refer to *a*, but to *a**3

How must the grammar be transformed, so that "-" refers to *a*?

Terminal Start Symbols of Nonterminals

Which terminal symbols can a nonterminal start with?

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

First(Factor) = ident, number, "("

First(Term) = First(Factor)
= ident, number, "("

First(Expr) = "+", "-", First(Term)
= "+", "-", ident, number, "("

Terminal Successors of Nonterminals

Which terminal symbols can follow after a nonterminal in the grammar?

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```

Follow(Expr) = ")", eof

Follow(Term) = "+", "-", Follow(Expr)
= "+", "-", ")", eof

Follow(Factor) = "*", "/", Follow(Term)
= "*", "/", "+", "-", ")", eof

Where does *Expr* occur on the right-hand side of a production? What terminal symbols can follow there?

Some Terminology

Alphabet

The set of terminal and nonterminal symbols of a grammar

String

A finite sequence of symbols from an alphabet.

Strings are denoted by greek letters (α, β, γ, ...)
e.g: α = ident + number
β = - Term + Factor * number

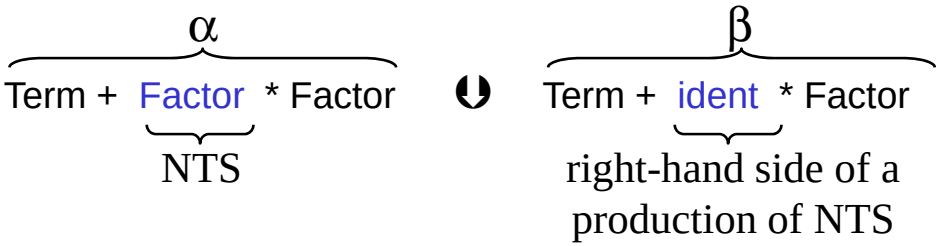
Empty String

The string that contains no symbol (denoted by ε).

Derivations and Reductions

Derivation

$\alpha \Rightarrow \beta$ (direct derivation)



$\alpha \Rightarrow^{*} \beta$ (indirect derivation) $\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$

$\alpha \Rightarrow_{L} \beta$ (left-canonical derivation) the <u>leftmost</u> NTS in α is derived first

$\alpha \Rightarrow_{R} \beta$ (right-canonical derivation) the <u>rightmost</u> NTS in α is derived first

Reduction

The converse of a derivation:
If the right-hand side of a production occurs in β it is replaced with the corresponding NTS

Deletability

A string α is called deletable, if it can be derived to the empty string.
α ⇩* ε

Example

```
A = B C.
B = [ b ].
C = c | d | .
```

B is deletable: B ⇩ ε

C is deletable: C ⇩ ε

A is deletable: A ⇩ B C ⇩ C ⇩ ε

More Terminology

Phrase

Any string that can be derived from a nonterminal symbol.

Term phrases: Factor
Factor * Factor
ident * Factor
...

Sentential form

Any string that can be derived from the start symbol of the grammar.

e.g.: Expr
Term + Term + Term
Term + Factor * ident + Term
...

Sentence

A sentential form that consists of terminal symbols only.

e.g.: ident * number + ident

Language (formal language)

The set of all sentences of a grammar (usually infinitely large).
e.g.: the C# language is the set of all valid C# programs

Recursion

A production is recursive if $A \Rightarrow^* \omega_1 A \omega_2$

Can be used to represent repetitions and nested structures

Direct recursion $A \Rightarrow \omega_1 A \omega_2$

Left recursion	A = b | A a.	A ⇩ A a ⇩ A a a ⇩ A a a a ⇩ b a a a a a ...
Right recursion	A = b | a A.	A ⇩ a A ⇩ a a A ⇩ a a a A ⇩ ... a a a a a b
Central recursion	A = b | "(" A ")".	A ⇩ (A) ⇩ ((A)) ⇩ (((A))) ⇩ (((... (b)...)))

Indirect recursion $A \Rightarrow^* \omega_1 A \omega_2$

Example

```
Expr   = Term { "+" Term }.
Term   = Factor { "*" Factor }.
Factor = id | "(" Expr ")".
```

Expr ⇩ Term ⇩ Factor ⇩ "(" Expr ")"

How to Remove Left Recursion

Left recursion cannot be handled in topdown syntax analysis

```
A = b | A a.
```

Both alternatives start with *b*.
The parser cannot decide which one to choose

Left recursion can be transformed to iteration

```
E = T | E "+" T.
```

What sentences can be derived?

```
T
T + T
T + T + T
...
```

From this one can deduce the iterative EBNF rule:

```
E = T { "+" T }.
```

1. Overview

Plain BNF Notation

terminal symbols	are written without quotes (ident, +, -)
nonterminal symbols	are written in angle brackets (<Expr>, <Term>)
sides of a production	are separated by ::=

BNF grammar for arithmetic expressions

```
<Expr>    ::=  <Sign> <Term>
<Expr>    ::=  <Expr> <Addop> <Term>

<Sign>    ::=  +
<Sign>    ::=  -
<Sign>    ::=

<Addop>   ::=  +
<Addop>   ::=  -

<Term>    ::=  <Factor>
<Term>    ::=  <Term> <Mulop> <Factor>

<Mulop>   ::=  *
<Mulop>   ::=  /

<Factor>  ::=  ident
<Factor>  ::=  number
<Factor>  ::=  ( <Expr> )
```

- Alternatives are transformed into separate productions
- Repetition must be expressed by recursion

Advantages

- fewer meta symbols (no |, (), [], {})
- it is easier to build a syntax tree

Disadvantages

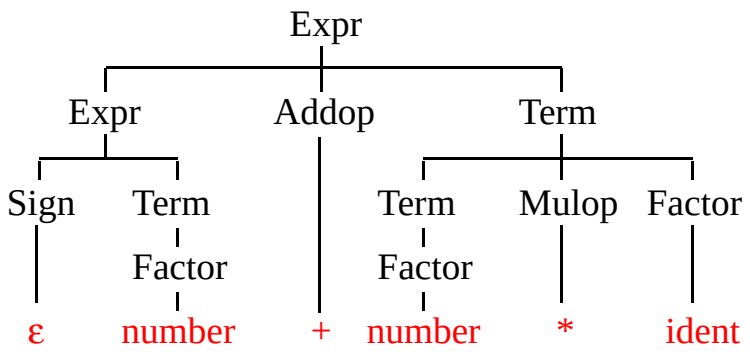
- more clumsy

Syntax Tree

Shows the structure of a particular sentence

e.g. for 10 + 3 * i

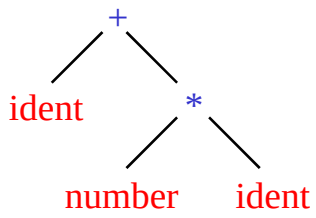
Concrete Syntax Tree (parse tree)



Would not be possible with EBNF because of [...] and {...}, e.g.:
Expr = [Sign] Term { Addop Term }.

Also reflects operator priorities: operators further down in the tree have a higher priority than operators further up in the tree.

Abstract Syntax Tree (leaves = operands, inner nodes = operators)



often used as an internal program representation; used for optimizations

Ambiguity

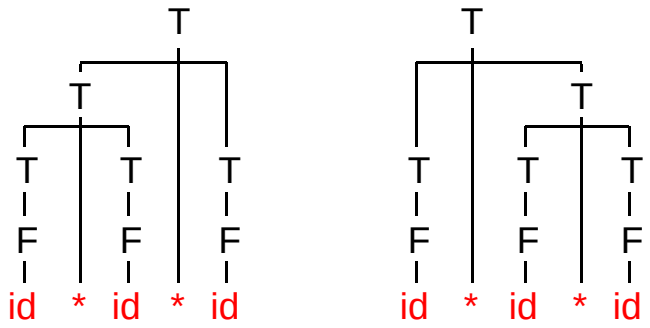
A grammar is ambiguous, if more than one syntax tree can be built for a given sentence.

Example

```
T = F | T "*" T.
F = id.
```

sentence: id * id * id

Two syntax trees can be built for this sentence:



Ambiguous grammars cause problems in syntax analysis!

Removing Ambiguity

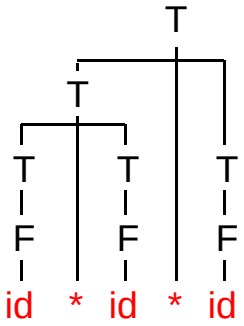
Example

```
T = F | T "*" T.
F = id.
```

Only the grammar is ambiguous, not the language.

The grammar can be transformed to

```
T = F | T "*" F.
F = id.
```



i.e. T has priority over F

only this syntax tree is possible

Even better: transformation to EBNF

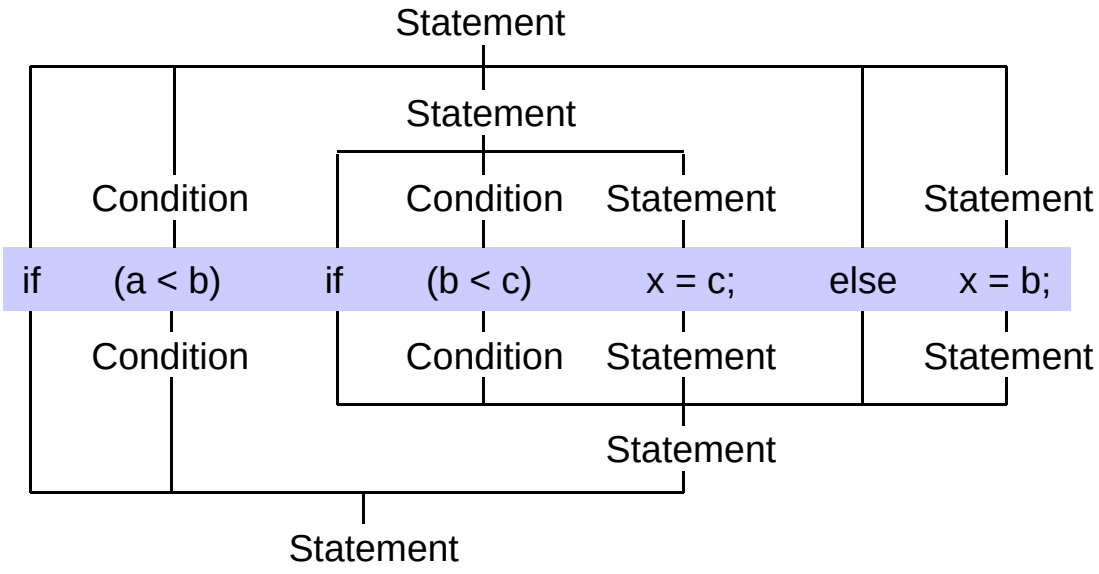
```
T = F { "*" F }.
F = id.
```

Inherent Ambiguity

There are languages which are inherently ambiguous

Example: *Dangling Else*

```
Statement =  Assignment
           |  "if" Condition Statement
           |  "if" Condition Statement "else" Statement
           |  ... .
```



There is no unambiguous grammar for this language!

C# solution

Always recognize the longest possible right-hand side of a production

- leads to the lower of the two syntax trees

1. Overview

Classification of Grammars

Due to Noam Chomsky (1956)

Grammars are sets of productions of the form α = β.

class 0 **Unrestricted grammars** (α and β arbitrary)

e.g: A = a A b | B c B.
aBc = d.
dB = bb.

A ⇩ aAb ⇩ aBcBb ⇩ dBb ⇩ bbb

Recognized by Turing machines

class 1 **Contex-sensitive grammars** (|α| ⇦ |β|)

e.g: a A = a b c.

Recognized by linear bounded automata

class 2 **Context-free grammars** (α = NT, β ≠ ε)

e.g: A = a b c.

Recognized by push-down automata

class 3 **Regular grammars** (α = NT, β = T | T NT)

e.g: A = b | b B.

Recognized by finite automata

Only these two classes are relevant in compiler construction

1. Overview

Sample Z# Program


```
class P
   const int size = 10;
   class Table {
      int[] pos;
      int[] neg;
   }
   Table val;
{
   void Main ()
      int x, i;
   {  /*---------- initialize val ----------*/
      val = new Table;
      val.pos = new int[size];
      val.neg = new int[size];
      i = 0;
      while (i < size) {
         val.pos[i] = 0; val.neg[i] = 0; i++;
      }
      /*---------- read values ----------*/
      read(x);
      while (-size < x && x < size) {
         if (0 <= x) { val.pos[x]++; }
         else { val.neg[-x]++; }
         read(x);
      }
   }
}
```

main program class; no separate compilation

inner classes (without methods)

global variables

local variables

Lexical Structure of Z#

Names ident = letter { letter | digit | '_' }.

Numbers number = digit { digit }. all numbers are of type *int*

Char constants charConst = '\'' char '\''. all character constants are of type *char* (may contain \r and \n)

no strings

Keywords

class						
if	else	while	read	write	return	break
void	const	new				

Operators

+	-	*	/	%	++	--
==	!=	>	>=	<	<=	
&&	||					
(	)	[	]	{	}	
=	;	,	.			

Comments /* ... */ may be nested

Types *int* *char* arrays classes

Syntactical Structure of Z# (1)

Programs

```
Program  =  "class" ident
            { ConstDecl | VarDecl | ClassDecl }
            "{" { MethodDecl } "}".
```

```
class P
   ... declarations ...
{  ... methods ...
}
```

Declarations

```
ConstDecl    =  "const" Type ident "=" ( number | charConst ) ";".
VarDecl      =  Type ident { "," ident } ";".
MethodDecl   =  (Type | "void") ident "(" [ FormPars ] ")" Block.

Type         =  ident [ "[" "]" ].
FormPars     =  Type ident { "," Type ident }.
```

only one-dimensional arrays

Syntactical Structure of Z# (2)

Statements

```
Block       =  "{" {Statement} "}".
Statement   =  Designator  (   "=" Expr ";"
                             |   "(" [ActPars] ")" ";"
                             |   "++" ";"
                             |   "--" ";"
                             )
            |  "if" "(" Condition ")" Block [ "else" Block ]
            |  "while" "(" Condition ")" Block
            |  "break" ";"
            |  "return" [ Expr ] ";"
            |  "read" "(" Designator ")" ";"
            |  "write" "(" Expr [ "," number ] ")" ";"
            |  ";".
ActPars     =  Expr { "," Expr }.
```

- input from *System.Console*
- output to *System.Console*

Syntactical Structure of Z# (3)

Expressions

```
Condition   =  CondTerm { "||" CondTerm }.
CondTerm    =  CondFact { "&&" CondFact }.
CondFact    =  Expr Relop Expr.
Relop       =  "==" | "!=" | ">" | ">=" | "<" | "<=".
```

```
Expr        =  [ "-" ] Term { Addop Term }.
Term        =  Factor { Mulop Factor }.
Factor      =  Designator [ "(" [ ActPars ] ")" ]
            |  number
            |  charConst
            |  "new" ident [ "[" Expr "]" ]
            |  "(" Expr ")".
Designator  =  ident [ "[" Expr "]" ] { "." ident [ "[" Expr "]" ] }.
Addop       =  "+" | "-".
Mulop       =  "*" | "/" | "%".
```

no constructors

2. Lexical Analysis

Tasks of a Scanner

1. **Delivers terminal symbols (tokens)**

character stream

token stream
(must end with EOF)

2. **Skips meaningless characters**
   - blanks
   - tabulator characters
   - end-of-line characters (CR, LF)
   - comments

Tokens have a syntactical structure, e.g.

```
ident =     letter { letter | digit }.
number =    digit { digit }.
if =        "i" "f".
eql =       "=" "=".
...
```

Why is scanning not part of parsing?

Why is Scanning not Part of Parsing?

It would make parsing more complicated
(e.g. difficult distinction between keywords and names)

```
Statement =  ident "=" Expr ";"
           |  "if" "(" Expr ")" ... .
```

One would have to write this as follows:

```
Statement =  "i" (   "f" "(" Expr ")" ...
                   |   notF {letter | digit} "=" Expr ";"
                   )
           |  notI {letter | digit} "=" Expr ";".
```

The scanner must eliminate blanks, tabs, end-of-line characters and comments
(these characters can occur anywhere => would lead to very complex grammars)

```
Statement =  "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ... .
Blank = " " | "\r" | "\n" | "\t" | Comment.
```

Tokens can be described with regular grammars
(simpler and more efficient than context-free grammars)

2. Lexical Analysis

Regular Grammars

Definition

A grammar is called regular if it can be described by productions of the form:

```
A = a.
A = b B.
```

a, b ∈ TS

A, B ∈ NTS

Example Grammar for names

```
Ident  =  letter
       |  letter Rest.
Rest   =  letter
       |  digit
       |  letter Rest
       |  digit Rest.
```

e.g., derivation of the name xy3

Ident ⇒ letter Rest ⇒ letter letter Rest ⇒ letter letter digit

Alternative definition

A grammar is called regular if it can be described by a single non-recursive EBNF production.

Example Grammar for names

```
Ident  =  letter { letter | digit }.
```

Examples

Can we transform the following grammar into a regular grammar?

```
E = T { "+" T }.
T = F { "*" F }.
F = id.
```

Can we transform the following grammar into a regular grammar?

```
E = F { "*" F }.
F = id | "(" E ")".
```

Limitations of Regular Grammars

Regular grammars cannot deal with *nested structures*
because they cannot handle *central recursion*!

But central recursion is important in most programming languages.

- nested expressions — Expr ❶ ... "(" Expr ")" ...
- nested statements — Statement ❶ "do" Statement "while" "(" Expr ")"
- nested classes — Class ❶ "class" "{" ... Class ... "}"

For productions like these we need context-free grammars.

But most lexical structures are regular

names	letter { letter | digit }
numbers	digit { digit }
strings	"\"" { noQuote } "\""
keywords	letter { letter }
operators	">" "="

Exception: nested comments

/* ..... /* ... */ ..... */

The scanner must treat them in a special way

Regular Expressions

Alternative notation for regular grammars

Definition

1. ε (the empty string) is a regular expression
2. A terminal symbol is a regular expression
3. If α and β are regular expressions the following expressions are also regular:

α β	
(α | β)	
(α)?	ε | α
(α)*	ε | α | αα | ααα | ...
(α)+	α | αα | ααα | ...

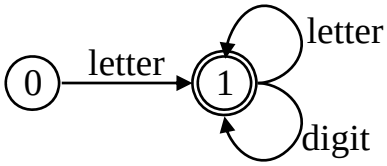
Examples

"w" "h" "i" "l" "e"	while
letter (letter | digit)*	names
digit+	numbers

Deterministic Finite Automaton (DFA)

Can be used to analyze regular languages

Example



final state

start state is always state 0 by convention

State transition function as a table

δ	letter	digit
s0	s1	error
s1	s1	s1

"finite", because δ can be written down explicitly

Definition

A deterministic finite automaton is a 5 tuple (S, I, δ, s0, F)

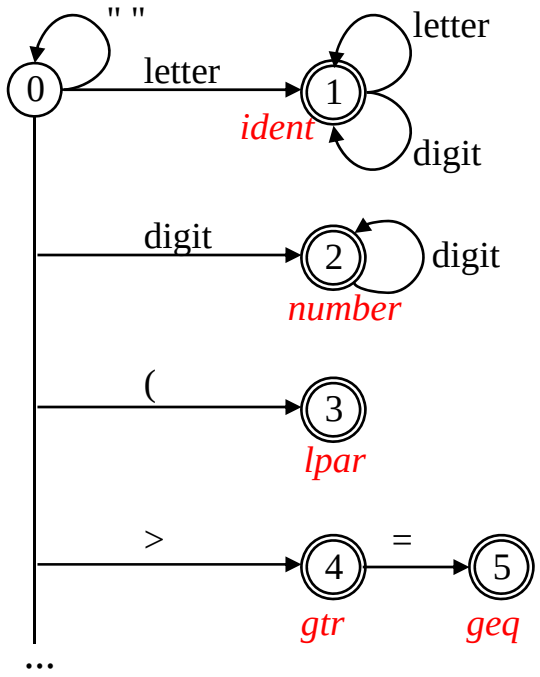
- S — set of states
- I — set of input symbols
- δ: S x I ✹ S — state transition function
- s0 — start state
- F — set of final states

The **language** recognized by a DFA is the set of all symbol sequences that lead from the start state into one of the final states

A DFA has recognized a sentence
- if it is in a final state
- and if the input is totally consumed or there is no possible transition with the next input symbol

The Scanner as a DFA

The scanner can be viewed as a big DFA



Example

input: max >= 30

s0 —m a x→ s1

- no transition with " " in s1
- *ident* recognized

s0 —> =→ s5

- skips blanks at the beginning
- does not stop in s4
- no transition with " " in s5
- *geq* recognized

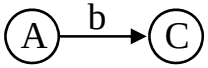
s0 —3 0→ s2

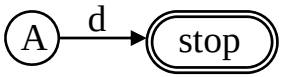
- skips blanks at the beginning
- no transition with " " in s2
- *number* recognized

After every recognized token the scanner starts in s0 again

Transformation: reg. grammar ★ DFA

A reg. grammar can be transformed into a DFA according to the following scheme

A = b C. ⊂ 

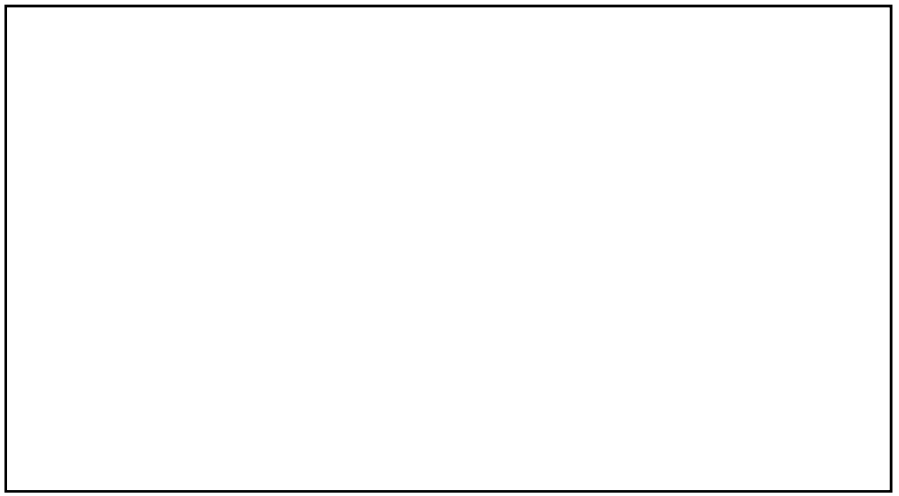
A = d. ⊂ 

Example

grammar

A = a B | b C | c.
B = b B | c.
C = a C | c.

automaton



Implementation of a DFA (Variant 1)

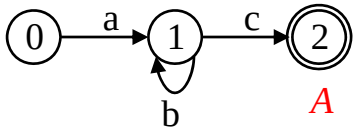
Implementation of δ as a matrix

```
int[,] delta = new int[maxStates, maxSymbols];
int lastState, state = 0;   // DFA starts in state 0
do {
    int sym = next symbol;
    lastState = state;
    state = delta[state, sym];
} while (state != undefined);
assert(lastState ∉ F);  // F is set of final states
return recognizedToken[lastState];
```

This is an example of a universal *table-driven algorithm*

Example for δ

A = a { b } c.

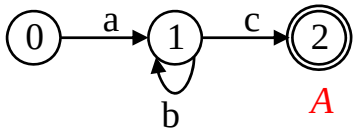


δ	a	b	c	
0	1	-	-	
1	-	1	2	
2	-	-	-	F

```
int[,] delta = { {1, -1, -1}, {-1, 1, 2}, {-1, -1, -1} };
```

This implementation would be too inefficient for a real scanner.

Implementation of a DFA (Variant 2)



Hard-coding the states in source code

```
char ch = read();
s0:   if (ch == 'a') { ch = read(); goto s1; }
      else goto err;
s1:   if (ch == 'b') { ch = read(); goto s1; }
      else if (ch == 'c') { ch = read(); goto s2; }
      else goto err;
s2:   return A;
err:  return errorToken;
```

In Java this is more tedious:

```
int state = 0;
loop:
   for (;;) {
      char ch = read();
      switch (state) {
         case 0:  if (ch == 'a') { state = 1; break; }
                  else break loop;
         case 1:  if (ch == 'b') { state = 1; break; }
                  else if (ch == 'c') { state = 2; break; }
                  else break loop;
         case 2: return A;
      }
   }
return errorToken;
```

2. Lexical Analysis

Scanner Interface


```
class Scanner {
   static void     Init (TextReader r) {...}
   static Token    Next () {...}
}
```

For efficiency reasons methods are static
(there is just one scanner per compiler)

Initializing the scanner

```
Scanner.Init(new StreamReader("myProg.zs"));
```

Reading the token stream

```
Token t;
for (;;) {
   t = Scanner.Next();
   ...
}
```

Tokens


```
class Token {
   int kind;       // token code
   int line;       // token line (for error messages)
   int col;        // token column (for error messages)
   int val;        // token value (for number and charCon)
   string str;     // token string (for numbers and identifiers)
}
```

Token codes for Z#

const int

error token

```
NONE = 0,
```

token classes

```
IDENT = 1,
NUMBER = 2,
CHARCONST = 3,
```

operators and special characters

```
PLUS   = 4,  /* + */      ASSIGN    = 17,/* = */
MINUS  = 5,  /* - */      PPLUS     = 18,/* ++ */
TIMES  = 6,  /* * */      MMINUS    = 19,/* -- */
SLASH  = 7,  /* / */      SEMICOLON = 20,/* ; */
REM    = 8,  /* % */      COMMA     = 21,/* , */
EQ     = 9,  /* == */     PERIOD    = 22,/* . */
GE     = 10, /* >= */     LPAR      = 23,/* ( */
GT     = 11, /* > */      RPAR      = 24,/* ) */
LE     = 12, /* <= */     LBRACK    = 25,/* [ */
LT     = 13, /* < */      RBRACK    = 26,/* ] */
NE     = 14, /* != */     LBRACE    = 27,/* { */
AND    = 15, /* && */     RBRACE    = 28,/* } */
OR     = 16, /* || */
```

keywords

```
BREAK  = 29,
CLASS  = 30,
CONST  = 31,
ELSE   = 32,
IF     = 33,
NEW    = 34,
READ   = 35,
RETURN = 36,
VOID   = 37,
WHILE  = 38,
WRITE  = 39,
```

end of file

```
EOF    = 40;
```

Scanner Implementation

Static variables in the scanner

```
static TextReader input;        // input stream
static char ch;                 // next input character (still unprocessed)
static int line, col;           // line and column number of the character ch
const int EOF = '\u0080';       // character that is returned at the end of the file
```

Init()

```
public static void Init (TextReader r) {
   input = r;
   line = 1; col = 0;
   NextCh();  // reads the first character into ch and increments col to 1
}
```

NextCh()

```
static void NextCh() {
   try {
      ch = (char) input.Read(); col++;
      if (ch == '\n') { line++; col = 0; }
      else if (ch == '\uffff') ch = EOF;
   } catch (IOException e) { ch = EOF; }
}
```

- *ch* = next input character
- returns *EOF* at the end of the file
- increments *line* and *col*

Method Next()

```
public static Token Next () {
   while (ch <= ' ') NextCh();  // skip blanks, tabs, eols
   Token t = new Token(); t.line = line, t.col = col;
   switch (ch) {
      case 'a': ... case 'z': case 'A': ... case 'Z': ReadName(t); break;
      case '0': case '1': ... case '9': ReadNumber(t); break;
      case ';': NextCh(); t.kind = Token.SEMICOLON; break;
      case '.': NextCh(); t.kind = Token.PERIOD; break;
      case EOF: t.kind = Token.EOF; break;  // no NextCh() any more
      ...
      case '=':    NextCh();
                   if (ch == '=') { NextCh(); t.kind = Token.EQ; }
                   else t.kind = Token.ASSIGN;
                   break;
      case '&':    NextCh();
                   if (ch == '&') { NextCh(); t.kind = Token.AND; }
                   else t.kind = NONE;
                   break;
      ...
      case '/':    NextCh();
                   if (ch == '/') {
                      do NextCh(); while (ch != '\n' && ch != EOF);
                      t = Next();  // call scanner recursively
                   } else t.kind = Token.SLASH;
                   break;
      default:     NextCh(); t.kind = Token.NONE; break;
   }
   return t;
} // ch holds the next character that is still unprocessed
```

names, keywords

numbers

simple tokens

composite tokens

comments

invalid character

Further Methods

static void ReadName (Token t)

- At the beginning *ch* holds the first letter of the name
- Reads further letters, digits and '_' and stores them in *t.str*
- Looks up the name in a keyword table (using hashing or binary search)
  if found: t.kind = *token number of the keyword*;
  otherwise: t.kind = Token.IDENT;
- At the end *ch* holds the first character after the name

static void ReadNumber (Token t)

- At the beginning *ch* holds the first digit of the number
- Reads further digits, storing them in *t.str*; then converts the digit string into a number and stores the value in *t.val.*
  if overflow: report an error
- t.kind = Token.NUMBER;
- At the end *ch* holds the first character after the number

Efficiency Considerations

Typical program size

about 1000 statements
- about 6000 tokens
- about 60000 characters

Scanning is one of the most time-consuming phases of a compiler (takes about 20-30% of the compilation time)

Touch every character as seldom as possible

therefore *ch* is global and not a parameter of N*extCh()*

For large input files it is a good idea to use buffered reading

```
Stream file = new FileStream("MyProg.zs");
Stream buf = new BufferedStream(file);
TextReader r = new StreamReader(buf);
Scanner.Init(r);
```

Does not pay off for small input files

3. Parsing

Context-Free Grammars

Problem

Regular Grammars cannot handle central recursion

```
E = x | "(" E ")".
```

For such cases we need context-free grammars

Definition

A grammar is called ***context-free*** (CFG) if all its productions have the following form:

A = α.

A ∈ NTS, α non-empty sequence of TS and NTS

In EBNF the right-hand side α can also contain the meta symbols |, (), [] a

Example

```
Expr = Term { ( "+" | "-" ) Term }.
Term = Factor { ( "*" | "/" ) Factor }.
Factor = id | "(" Expr ")".
```

indirect central recursion

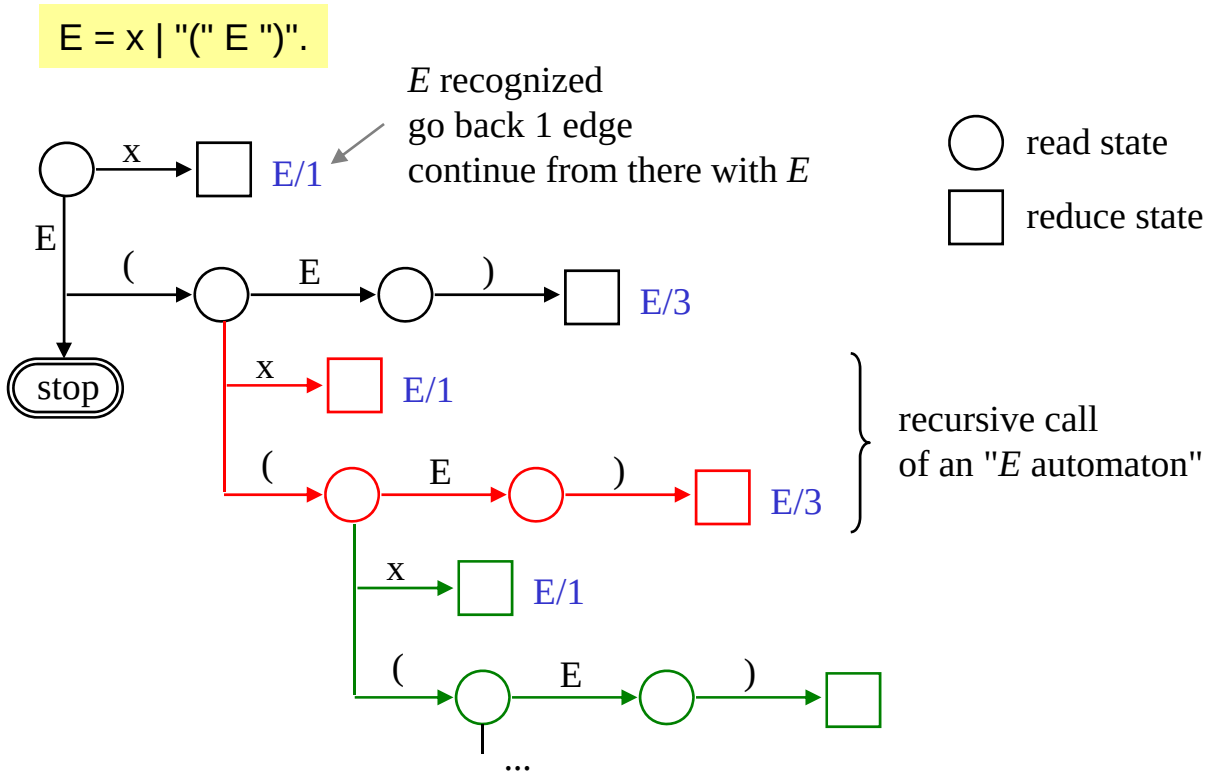
Context-free grammars can be recognized by ***push-down automata***

Push-Down Automaton (PDA)

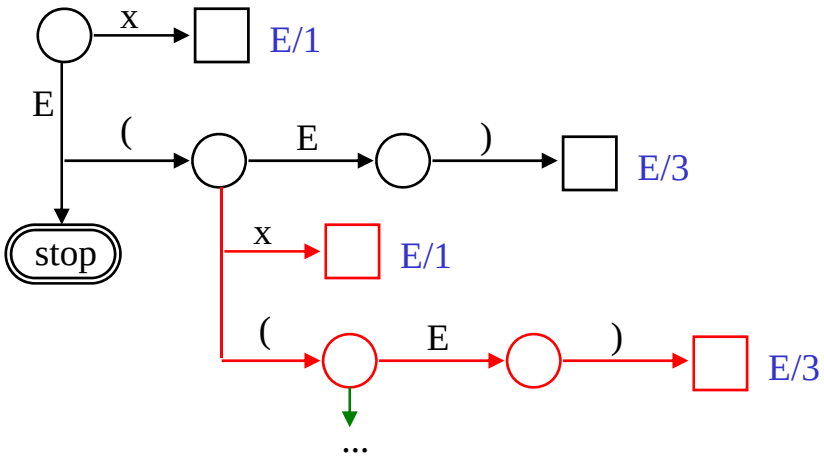
Characteristics

- Allows transitions with terminal symbols and nonterminal symbols
- Uses a stack to remember the visited states

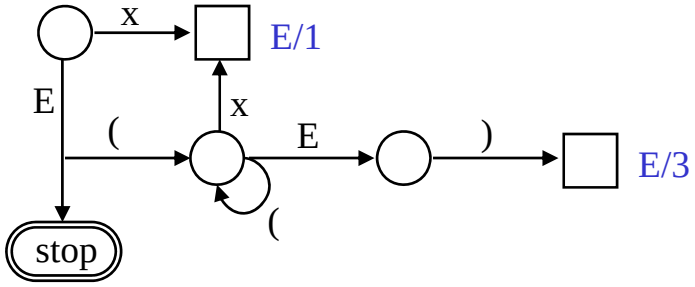
Example



Push-Down Automaton (cont.)



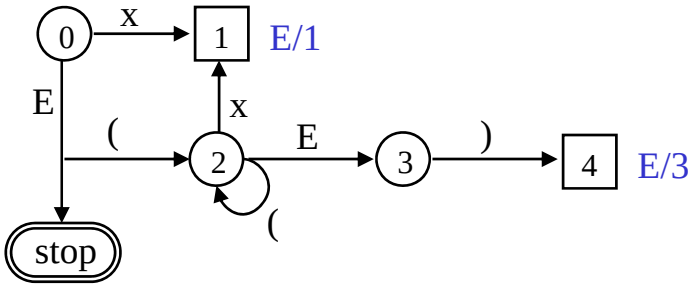
Can be simplified to



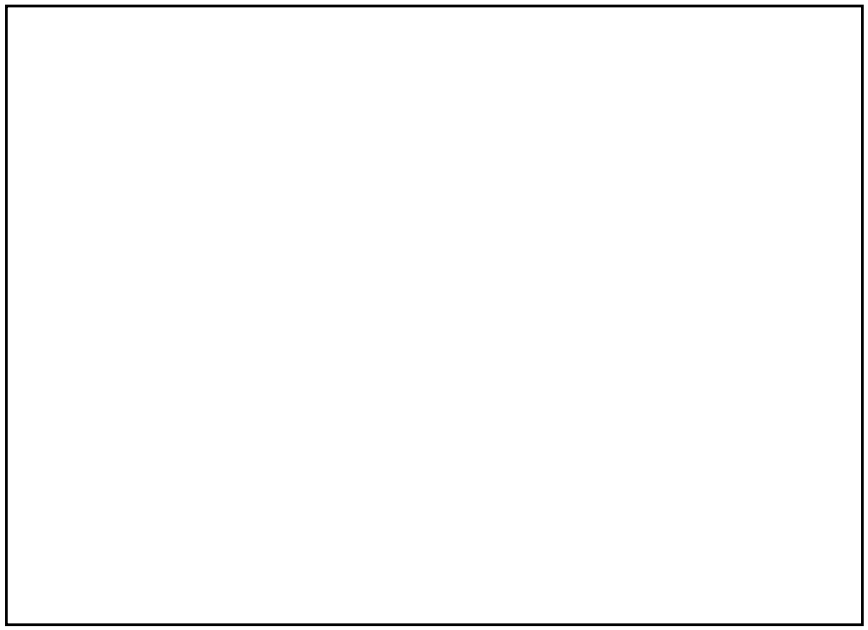
Needs a stack in order to find its way back through the visited states

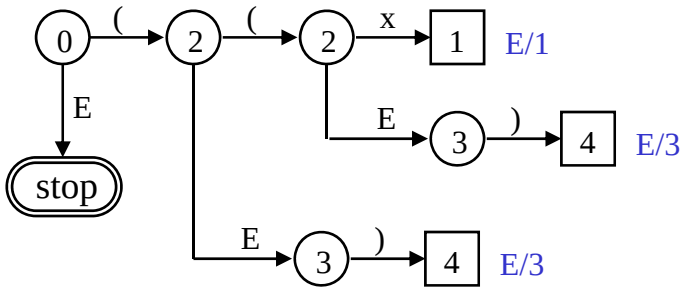
How a PDA Works

Example: input is ((x))



Visited states are stored in a stack





Limitations of Context-Free Grammars

CFGs cannot express *context conditions*

For example:

- ***Every name must be declared before it is used***
  The declaration belongs to the context of the use; the statement
  ```
  x = 3;
  ```
  may be right or wrong, depending on its context
- ***The operands of an expression must have compatible types***
  Types are specified in the declarations, which belong to the context of the use

Possible solutions

- ***Use context-sensitive grammars***
  too complicated
- ***Check context conditions during semantic analysis***
  i.e. the syntax allows sentences for which the context conditions do not hold
  ```
  int x = "three";        syntactically correct
                          semantically wrong
  ```
  The error is detected during semantic analysis (not during syntax analysis).

Context Conditions

Semantic constraints that are specified for every production

For example in Z#

```
Statement = Designator "=" Expr ";".
```

- *Designator* must be a variable, an array element or an object field.
- The type of *Expr* must be assignment compatible with the type of *Designator*.

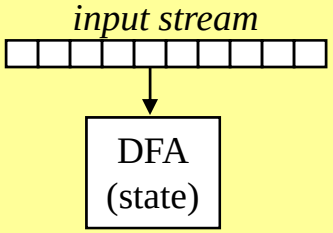
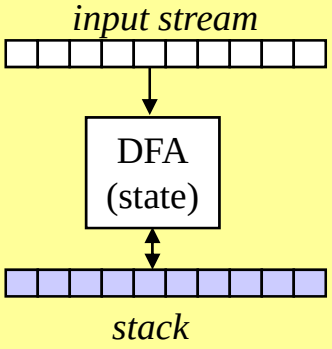
```
Factor = "new" ident "[" Expr "]".
```

- *ident* must denote a type.
- The type of *Expr* must be *int*.

Designator_1 = Designator_2 "[" Expr "]".

- The type of $Designator_2$ must be an array type.
- The type of *Expr* must be *int*.

Regular versus Context-free Grammars

	Regular Grammars	**Context-free Grammars**
Used for	Scanning	Parsing
Recogized by	**DFA** (no stack) 	**PDA** (stack) 
Productions	A = a | b C.	A = α.
Problems	nested language constructs	context-sensitive constructs (e.g. type checks, ...)

3. Parsing

Recursive Descent Parsing

- Top-down parsing technique
- The syntax tree is build from the start symbol to the sentence (top-down)

Example

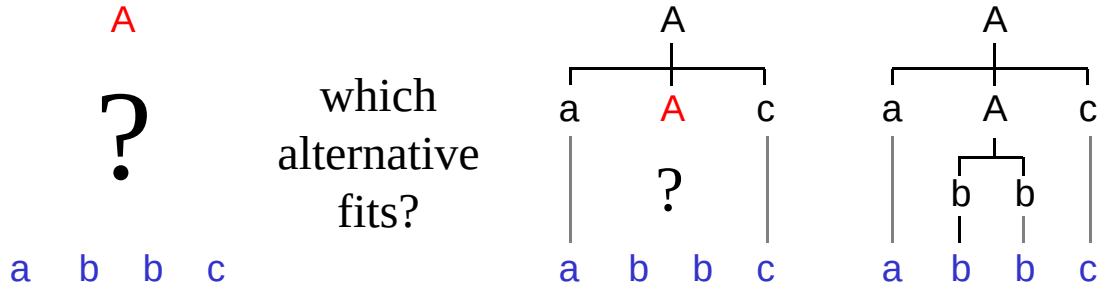
grammar

A = a A c | b b.

input

a b b c

start symbol



input

The correct alternative is selected using ...

- the lookahead token from the input stream
- the terminal start symbols of the alternatives

Static Variables of the Parser

Lookahead token

At any moment the parser knows the next input token

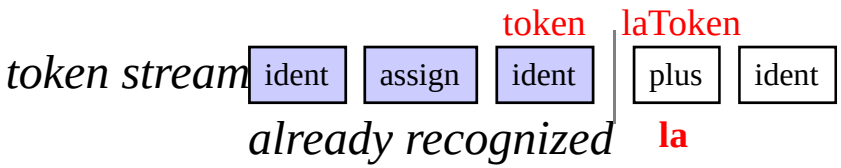
```
static int la;          // token number of the lookahead token
```

It remembers two input tokens (for semantic processing)

```
static Token token;     // most recently recognized token
static Token laToken;   // lookahead token (still unrecognized)
```

These variables are set in the method *Scan()*

```
static void Scan () {
   token = laToken;
   laToken = Scanner.Next();
   la = laToken.kind;
}
```



Scan() is called at the beginning of parsing ➧ first token is in *la*

How to Parse Terminal Symbols

Pattern

symbol to be parsed: a

parsing action: Check(a);

Needs the following auxiliary methods

```
static void Check (int expected) {
    if (la == expected) Scan();  // recognized => read ahead
    else Error( Token.names[expected] + " expected"  );
}
```

```
public static void Error (string msg) {
    Console.WriteLine("– line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
    throw new Exception("Panic Mode");  // for a better solution see later
}
```

in class *Token*:

```
public static string[] names = {"?", "identifier", "number", ..., "+", "-", ...};
```

ordered by token codes

The names of the terminal symbols are declared as constants in class *Token*

```
public const int  NONE = 0,
                  IDENT = 1, NUMBER = 2, ...,
                  PLUS = 4, MINUS = 5, ... ;
```

How to Parse Nonterminal Symbols

Pattern

symbol to be parsed: A

parsing action: A(); // call of the parsing method A

Every nonterminal symbol is recognized by a parsing method with the same name

```
private static void A() {
    ... parsing actions for the right-hand side of A ...
}
```

Initialization of the Z# parser

```
public static void Parse () {
    Scan();                 // initializes token, laToken and la
    Program();              // calls the parsing method of the start symbol
    Check(Token.EOF);       // at the end the input must be empty
}
```

How to Parse Sequences

Pattern

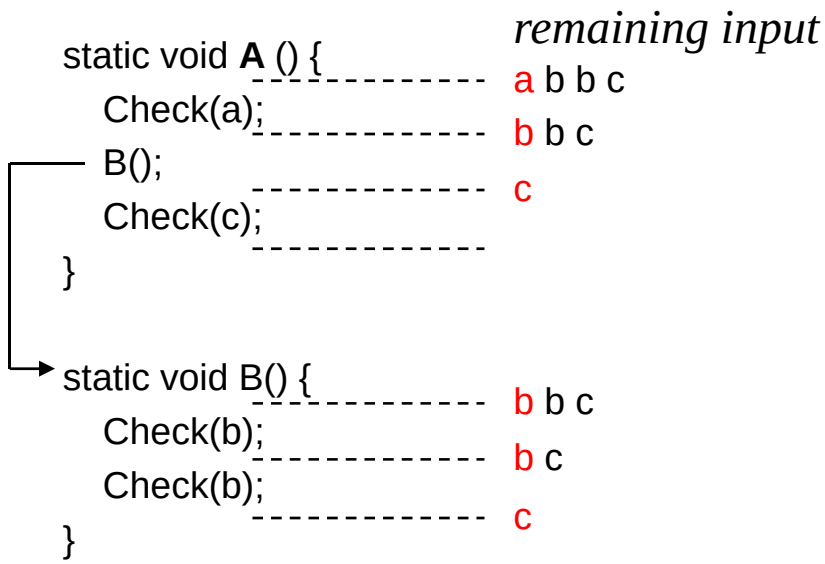
production: A = a B c.

parsing method:

```
static void A () {
   // la contains a terminal start symbol of A
   Check(a);
   B();
   Check(c);
   // la contains a follower of A
}
```

Simulation

A = a B c.
B = b b.



How to Parse Alternatives

Pattern $\alpha \mid \beta \mid \gamma$ α, β, γ are arbitrary EBNF expressions

Parsing action

```
if (la ∉ First(α)) { ... parse α ... }
else if (la ∉ First(β)) { ... parse β ... }
else if (la ∉ First(γ)) { ... parse γ ... }
else Error("...");  // find a meaninful error message
```

Example

```
A = a B | B b.
B = c | d.
```

First(aB) = {a}
First(Bb) = First(B) = {c, d}

```
static void A () {
    if (la == a) {
        Check(a);
        B();
    } else if (la == c || la == d) {
        B();
        Check(b);
    } else Error ("invalid start of A");
}
```

How to Parse EBNF Options

Pattern [α] α is an arbitrary EBNF expression

Parsing action if (la ∉ First(α)) { ... *parse* α ... } // no error branch!

Example

```
A = [ a b ] c.
```

```
static void A () {
    if (la == a) {
        Check(a);
        Check(b);
    }
    Check(c);
}
```

Example: parse a b c
parse c

How to Parse EBNF Iterations

Pattern { α } α is an arbitrary EBNF expression

Parsing action while (la ∉ First(α)) { ... *parse* α ... }

Example

```
A = a { B } b.
B = c | d.
```

```
static void A () {
   Check(a);
   while (la == c || la == d) B();
   Check(b);
}
```

Example: parse a c d c b
parse a b

alternatively ...

```
static void A () {
   Check(a);
   while (la != b && la != Token.EOF) B();
   check(b);
}
```

without EOF: danger of an infinite loop, if *b* is missing in the input

How to Deal with Large First Sets

If the set has more than 4 elements: use class *BitArray*

example: First(A) = {a, b, c, d, e}
First(B) = {f, g, h, i, j}

The First sets are initialized at the beginning of the program

```
using System.Collections;
static BitArray firstA = new BitArray(Token.names.Length);
firstA[a] = true; firstA[b] = true; firstA[c] = true; firstA[d] = true; firstA[e] = true;
static BitArray firstB = new BitArray(Token.names.Length);
firstB[f] = true; firstB[g] = true; firstB[h] = true; firstB[i] = true; firstB[j] = true;
```

Set test

```
C = A | B.
```

```
static void C () {
    if (firstA[la]) A();
    else if (firstB[la]) B();
    else Error("invalid C");
}
```

How to Deal with Large First Sets

If the set has less than 5 elements: use explicit checks (which is faster)

e.g.: First(A) = {a, b, c}

```
if (la == a || la == b || la == c) ...
```

If the set is an interval: use an interval test

```
if (a <= la && la <= c) ...
```

Token codes can often be chosen so that frequently checked sets form intervals

Example

First(A) = { a, c, d }
First(B) = { a, d }
First(C) = { b, e }

```
const int
  a = 0,
  d = 1,
  c = 2,
  b = 3,
  e = 4,
```

First(B)
First(A)
First(C)

Optimizations

Avoiding multiple checks

```
A = a | b.
```

```
static void A () {
    if (la == a) Scan();  // no Check(a);
    else if (la == b) Scan();
    else Error("invalid A");
}
```

```
A = { a | B d }.
B = b | c.
```

```
static void A () {
    while (la == a || la == b || la == c) {
        if (la == a) Scan();
        else { // no check any more
            B(); Check(d);
        } // no error case
    }
}
```

More efficient scheme for parsing alternatives in an iteration

```
A = { a | B d }.
```

```
static void A () {
    for (;;) {
        if (la == a) Scan();
        else if (la == b || la == c) { B(); Check(d); }
        else break;
    }
}
```

Optimizations

Frequent iteration pattern

α { separator α }

```
for (;;) {
    ... parse α ...
    if (la == separator) Scan(); else break;
}
```

Example

ident { "," ident }

```
for (;;) {
    Check(ident);
    if (la == Token.COMMA) Scan(); else break;
}
```

input e.g.: a , b , c :

Computing Terminal Start Symbols Correctly

Grammar

```
A = B a.
B = { b } c
  | [ d ]
  | e.
```

terminal start symbols

b and *c*

d and *a* (!)

e

```
C = D e
  | f.
D = { d }.
```

d and *e* (*D* is deletable!)

f

Parsing methods

```
static void A () {
   B(); Check(a);
}
```

```
static void B () {
   if (la == b || la == c) {
      while (la == b) Scan();
      Check(c);
   } else if (la == d || la == a) {
      if (la == d) Scan();
   } else if (la == e) {
      Scan();
   } else Error("invalid B");
}
```

```
static void C () {
   if (la == d || la == e) {
      D(); Check(e);
   } else if (la == f) {
      Scan();
   } else Error("invalid C");
}
```

```
static void D () {
   while (la == d) Scan();
}
```

Recursive Descent and the Syntax Tree

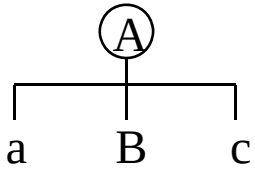
The syntax tree is only built implicitly

- it is denoted by the methods that are currently active
- i.e. by the productions that are currently under examination

Example A = a B c.
B = d e.

call *A()*

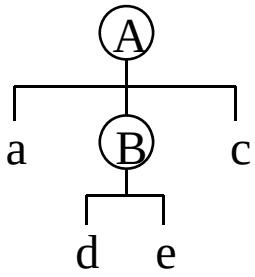
```
static void A () {
    Check(a); B(); Check(c);
}
```



A in process

recognize *a*
call *B()*

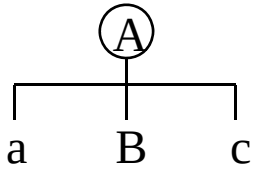
```
static void B () {
    Check(d); Check(e);
}
```



A in process
B in process

"stack"

recognize *d* and *e*
return from *B()*



A in process

3. Parsing

LL(1) Property

Precondition for recursive descent parsing

LL(1) ... can be analyzed from **L**eft to right
with **L**eft-canonical derivations (leftmost NTS is derived first)
and **1** lookahead symbol

Definition

1. A grammar is LL(1) if all its productions are LL(1).
2. A production is LL(1) if for all its alternative
   $\alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
   the following condition holds:
   $First(\alpha_i) \cap First(\alpha_j) = \{\}$ (for any *i* and *j*)

In other words

- The terminal start symbols of all alternatives of a production must be pairwise disjoint.
- The parser must always be able to select one of the alternatives by looking at the lookahead token.

How to Remove LL(1) Conflicts

Factorization

```
IfStatement =  "if" "(" Expr ")" Statement
             |  "if" "(" Expr ")" Statement "else" Statement.
```

Extract common start sequences

```
IfStatement =  "if" "(" Expr ")" Statement (
                                            | "else" Statement
                                            ).
```

... or in EBNF

```
IfStatement =  "if" "(" Expr ")" Statement [ "else" Statement ].
```

Sometimes nonterminal symbols must be inlined before factorization

```
Statement   =  Designator "=" Expr ";"
             |  ident "(" [ ActualParameters ] ")" ";".
Designator  =  ident { "." ident }.
```

Inline *Designator* in *Statement*

```
Statement   =  ident { "." ident } "=" Expr ";"
             |  ident "(" [ ActualParameters ] ")" ";".
```

then factorize

```
Statement   =  ident  (   { "." ident } "=" Expr ";"
                       |   "(" [ ActualParameters ] ")" ";"
                       ).
```

How to Remove Left Recursion

Left recursion is always an LL(1) conflict

For example

```
IdentList = ident | IdentList "," ident.
```

generates the following phrases

```
ident
ident "," ident
ident "," ident "," ident
...
```

can always be replaced by iteration

```
IdentList = ident { "," ident }.
```

Hidden LL(1) Conflicts

EBNF options and iterations are hidden alternatives

A = [α] β. ⊂ A = α β | β. α and β are arbitrary EBNF expressions

Rules

A = [α] β.	First(α) ∩ First(β) must be {}
A = { α } β.	First(α) ∩ First(β) must be {}

A = α [β].	First(β) ∩ Follow(A) must be {}
A = α { β }.	First(β) ∩ Follow(A) must be {}

A = α | .	First(α) ∩ Follow(A) must be {}

Removing Hidden LL(1) Conflicts


```
Name = [ ident "." ] ident.
```

Where is the conflict and how can it be removed?

```
Prog = Declarations ";" Statements.
Declarations = D { ";" D }.
```

Where is the conflict and how can it be removed?

Dangling Else

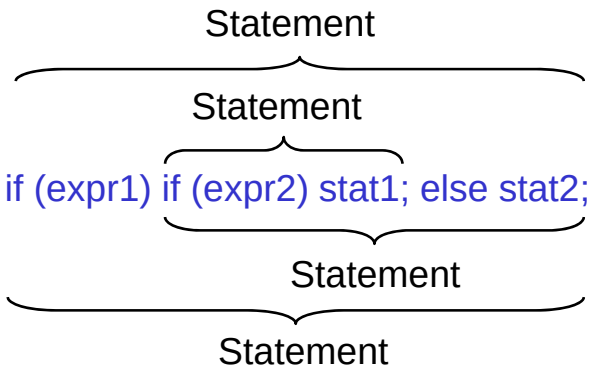
If statement in C#

```
Statement =  "if" "(" Expr ")" Statement [ "else" Statement ]
           |  ... .
```

This is an LL(1) conflict!

First("else" Statement) ⋂ Follow(Statement) = {"else"}

It is even an ambiguity which cannot be removed



We can build 2 different syntax trees!

Can We Ignore LL(1) Conflicts?

An LL(1) conflict is only a warning

The parser selects the first matching alternative

```
A = a b c
  | a d.
```

← if the lookahead token is *a* the parser selects this alternative

Example: Dangling Else

```
Statement = "if" "(" Expr ")" Statement [ "else" Statement ]
          | ... .
```

If the lookahead token is "else" here
the parser starts parsing the option;
i.e. the "else" belongs to the innermost "if"

```
if (expr1) if (expr2) stat1; else stat2;
                      ^^^^^^^^^^^^^^^^^
                        Statement
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             Statement
```

Luckily this is what we want here.

Other Requirements for a Grammar

(Preconditions for Parsing)

Completeness

For every NTS there must be a production

```
A = a B C .
B = b b .
```

error!
no production for *C*

Derivability

Every NTS must be derivable (directly or indirectly) into a string of terminal symbols

```
A = a B | c .
B = b B .
```

error!
B cannot be derived into a string of terminal symbols

Non-circularity

A NTS must not be derivable (directly or indirectly) into itself ($A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow A$)

```
A = a b | B .
B = b b | A .
```

error!
this grammar is circular because of $A \Rightarrow B \Rightarrow A$

3. Parsing

Goals of Syntax Error Handling

Requirements

1. The parser should detect as many errors as possible in a single compilation
2. The parser should never crash (even in the case of abstruse errors)
3. Error handling should not slow down error-free parsing
4. Error handling should not inflate the parser code

Error handling techniques for recursive descent parsing

- Error handling with "panic mode"
- Error handling with "general anchors"
- Error handling with "special anchors"

Panic Mode

The parser gives up after the first error

```
static void Error (string msg) {
    Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
    throw new Exception("Panic Mode - exiting after first error");

}
```

Advantages

- cheap
- sufficient for small command languages or for interpreters

Disadvantages

- not appropriate for production-quality compilers

Error Handling with "General Anchors"

Example

expected input: a b c d ...
real input: a x y d ...

Recovery (synchronize the remaining input with the grammar)

1. **Find "anchor tokens" with which the parser can continue after the error.**

   What are the tokens which the parser can continue with in the above situation?
   c successor of *b* (which was expected at the error position)
   d successor of *b c*
   ...
   Anchors at this position are {*c*, *d*, ...}
2. **Skip input tokens until an achor is found.**
   x and *y* are skipped here, but *d* is an anchor; the parser can continue with it.

3. **Drive the parser to the position in the grammar from where it can continue.**

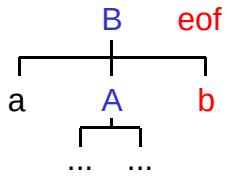
Computing Anchors

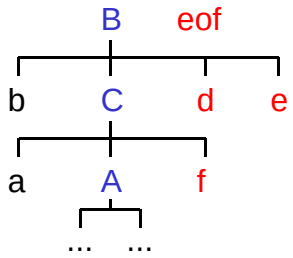
Every parsing method of a nonterminal *A* gets the current successors of *A* as parameters

```
static void A (BitArray sux) {
    ...
}
```

sux ... successors of all NTS, that are currently in process

Depending on the current context sux_A can denote different sets





$sux_A = \{b, eof\}$

$sux_A = \{f, d, e, eof\}$

sux always contains *eof* (the successor of the start symbol)

Handling Terminal Symbols

Grammar

A = ... a s_1 s_2 ... s_n .

s_i ∉ TS ⇨ NTS

Parsing action

check(a, sux_A ⇨ First(s_1) ⇨ First(s_2) ⇨ ... ⇨ First(s_n));

can be precomputed at compile time

must be computed at run time

```
static void Check (int expected, BitArray sux) {...}
```

Example

A = a b c.

```
static void A (BitArray sux) {
    Check(a, Add(sux, fs1));
    Check(b, Add(sux, fs2));
    Check(c, sux);
}
```

```
static BitArray fs1 = new BitArray();
fs1[b] = true; fs1[c] = true;
```

computed at the beginning of the program

```
static BitArray Add (BitArray a, BitArray b) {
    BitArray c = (BitArray) a.Clone();
    c[b] = true;
    return c;
}
```

Handling Nonterminal Symbols

Grammar

A = ... B s_1 s_2 ... s_n .

Parsing action

B(sux_A ⇗ First(s_1) ⇗ First(s_2) ⇗ ... ⇗ First(s_n));

Example

A = a B c.
B = b b.

```
static void A (BitArray sux) {
    Check(a, Add(sux, fs3));
    B(Add(sux, fs4));
    Check(c, sux);
}
```

fs3 = {b, c}
fs4 = {c}

```
static void B (BitArray sux) {
    Check(b, Add(sux, fs5));
    Check(b, sux);
}
```

fs5 = {b}

The parsing method for the start symbol *S* is called as *S(fs0);* where *fs0* = {*eof*}

Skipping Invalid Input Tokens

Errors are detected in *check()*

```
static void Check (int expected, BitArray sux) {
   if (la == expected) Scan();
   else Error(Token.names[expected] + " expected", sux);
}
```

After printing an error message input tokens are skipped until an anchor occurs

```
static void Error (string msg, BitArray sux) {
   Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
   errors++;
   while (!sux[la]) Scan();  // while (la ∉ sux) Scan();
   // la ∉ sux
}
```

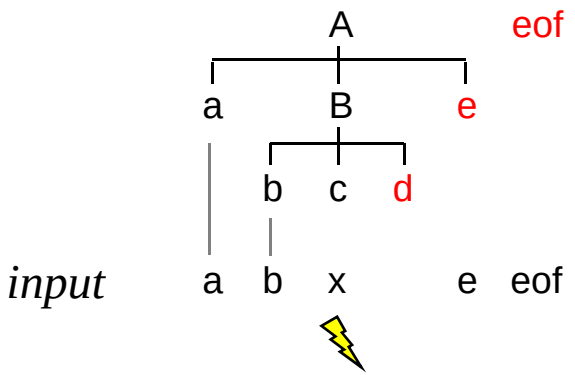
```
static int errors = 0;  // number of syntax errors detected
```

Synchronizing With the Grammar

Example

```
A = a B e.
B = b c d.
```

```
static void A (BitArray sux) {
    Check(a, Add(sux, fs1));
    B(Add(sux, fs2));
    Check(e, sux);
}
static void B (BitArray sux) {
    Check(b, Add(sux, fs3));
    Check(c, Add(sux, fs4));
    Check(d, sux);
}
```



$sux_A = \{eof\}$

$sux_B = \{e, eof\}$

the error is detected here; anchors = {*d*, *e*, *eof*}

1. *x* is skipped; *la* == *e* (Ø anchors)
2. parser continues: *Check(d, sux);*
3. detects an error again; anchors = {*e*, *eof*}
4. nothing is skipped, because *la* == *e* (Ø anchors)
5. parser returns from *B()* and does *Check(e, sux);*
6. recovery succeeded!

After the error the parser "jolts ahead" until it gets to a point in the grammar where the found anchor token is valid.

Suppressing Spurious Error Messages

During error recovery the parser produces spurious error messages

Solved by a simple heuristics

If less than 3 tokens were recognized correctly since the last error, the parser assumes that the new error is a spurious error. Spurious errors are not reported.

```
static int errDist = 3;  // next error should be reported
```

```
static void Scan () {
   ...
   errDist++;  // one more token recognized
}
```

```
public static void Error (string msg, BitArray sux) {
   if (errDist >= 3) {
      Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
      errors++;
   }
   while (!sux[la]) Scan();
   errDist = 0;  // counting is restarted
}
```

Handling Alternatives

A = α | β | γ .

α, β, γ are arbitrary EBNF expressions

```
static void A (BitArray sux) {
   // the error check is already done here so that the parser can synchronize with
   // the starts of the alternatives in case of an error
   if (la ∉ (First(α) ⊛ First(β) ⊛ First(γ)))
      Error("invalid A", sux ⊛ First(α) ⊛ First(β) ⊛ First(γ));
   // la matches one of the alternatives or is a legal successor of A
   if (la ∈ First(α)) ... parse α ...
   else if (la ∈ First(β)) ... parse β ...
   else ... parse γ ... // no error check here; any errors have already been reported
}
```

First(α) ⊛ First(β) ⊛ First(γ) can be precomputed at compile time

sux ⊛ ... must be computed at run time

Handling EBNF Options and Iterations

Options

A = [α] β.

```
static void A (BitArray sux) {
   // error check already done here, so that the parser can
   // synchronize with the start of α in case of an error
   if (la ∉ (First(α) ⊛ First(β)))
      Error("...", sux ⊛ First(α) ⊛ First(β));
   // la matches α or β or is a successor of A
   if (la ∈ First(α)) ... parse α ...;
   ... parse β ...
}
```

Iterations

A = { α } β.

```
static void A (BitArray sux) {
   for (;;) {
      // the loop is entered even if la ∉ First(α)
      if (la ∈ First(α)) ... parse α ...;                  // correct case 1
      else if (la ∈ First(β) ⊛ sux) break;                 // correct case 2
      else Error("...", sux ⊛ First(α) ⊛ First(β)); // error case
   }
   ... parse β ...
}
```

Example


```
A = a B | b {c d}.
B = [b] d.
```

```
static void A (BitArray sux) {
    if (la != a && la != b)
        Error("invalid A", Add(sux, fs1));  // fs1 = {a, b}
    if (la == a) {
        Scan(); B(sux);
    } else if (la == b) {
        Scan();
        for (;;) {
            if (la == c) {
                Scan();
                Check(d, Add(sux, fs2));  // fs2 = {c}
            } else if (sux[la]) {
                break;
            } else {
                Error("c expected", Add(sux, fs2));
            }
        }
    }
}
```

```
static void B (BitArray sux) {
    if (la != b && la != d)
        Error("invalid B", Add(sux, fs3));  // fs3 = {b, d}
    if (la == b) Scan();
    Check(d, sux);
}
```

Assessment

Error handling with general anchors

Advantage

+ systematically applicable

Disadvantages

- slows down error-free parsing
- inflates the parser code
- complicated

Error Handling With Special Anchors

Error handling is only done at particularly "safe" positions

i.e. at positions that start with keywords which do not occur at any other position in the gra

For example

anchor sets

- start of Statement: if, while, do, ...
- start of Declaration: public, static, void, ...

Problem: *ident* can occur at both positions!

ident is not a safe anchor ↯ omit it from the anchor set

Code that has to be inserted at the synchronization points

```
...                                  anchor set at this synchronization point
if (la ∉ expectedSymbols) {
   Error("...");  // no successor sets; no skipping of tokens in Error()
   while (la ∉ (expectedSymbols ⇧ {eof})) Scan();
}
...                                  in order not to get into an endless loop
```

- No successor sets have to be passed to parsing methods
- Anchor sets can be computed at compile time
- After an error the parser "jolts ahead" to the next synchronization point

Example

Synchronization at the start of Statement

```
static void Statement () {
   if (!firstStat[la]) {
      Error("invalid start of statement");
      while (!firstStat[la] && la != Token.EOF) Scan();
   }
   if (la == Token.IF) { Scan();
      Check(Token.LPAR);
      Conditions();
      Check(Token.RPAR);
      Statement();
      if (la == Token.ELSE) { Scan(); Statement(); }
   } else if (la == Token.WHILE) {
      ...
}
```

```
static BitArray firstStat = new BitArray();
firstStat[Token.WHILE] = true;
firstStat[Token.IF] = true;
...
```

the rest of the parser remains unchanged (as if there were no error handling)

No Synchronization in *Error()*

```
public static void Error (string msg) {
   if (errDist >= 3) {
      Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
      errors++;
   }
   errDist = 0;
}
```

heuristics with *errDist* can also be applied here

Example of a Recovery


```
static void Statement () {
   if (!firstStat[la]) {
      Error("invalid start of statement");
      while (!firstStat[la] && la != Token.EOF) Scan();
   }
   if (la == Token.IF) { Scan();
      Check(Token.LPAR);
      Condition();
      Check(Token.RPAR);
      Statement();
      if (la == Token.ELSE) { Scan(); Statement(); }
   ...
}
```

```
static void Check (int expected) {
   if (la == expected) Scan();
   else Error(...);
}
```

```
public static void Error (string msg) {
   if (errDist >= 3) {
      Console.WriteLine(...);
      errors++;
   }
   errDist = 0;
}
```

erroneous input: if a > b then max = a;

la	*action*	
IF	Scan();	IF ∉ *firstStatement* ⇒ ok
a	Check(LPAR);	error message: (expected
	Condition();	recognizes a > b
THEN	check(RPAR);	error message:) expected
	Statement();	THEN does not match ⇒ error, but no error message
		THEN is skipped; synchronization with *ident* (if in *firstStat*)
max		

Synchronization at the Start of an Iteration

For example

```
Block = "{" { Statement } "}".
```

Standard pattern in this case

```
static void Block () {
   Check(Token.LBRACE);
   while (firstStat[la])
      Statement();
   Check(Token.RBRACE);
}
```

Problem: If the next token does not match *Statement* the loop is not executed.
Synchronization point in *Statement* is never reached.

Synchronization at the Start of an Iteration

For example

```
Block = "{" { Statement } "}".
```

Better to synchronize at the beginning of the iteration

```
static void Block() {
   Check(Token.LBRACE);
   for (;;) {
      if (la ∉ First(Statement)) Statement();      // correct case 1
      else if (la ∉ {rbrace, eof}) break;          // correct case 2
      else {                                       // error case
         Error("invalid start of Statement");
         do Scan(); while (la ∉ (First(Statement) ⇾ {rbrace, eof}));
      }
   }
   Check(Token.RBRACE);
}
```

No synchronization in *Statement()* any more

```
static void Statement () {
   if (la == Token.IF) { Scan(); ...
}
```

Assessment

Error handling with special anchors

Advantages

+ does not slow down error-free parsing
+ does not inflate the parser code
+ simple

Disadvantage

- needs experience and "tuning"

4. Semantic Processing and Attribute Grammars

Semantic Processing

The parser checks only the *syntactic* correctness of a program

Tasks of semantic processing

- **Symbol table handling**
  - Maintaining information about declared names
  - Maintaining information about types
  - Maintaining scopes
- **Checking context conditions**
  - Scoping rules
  - Type checking
- **Invocation of code generation routines**

Semantic actions are integrated into the parser
and are described with *attribute grammars*

Semantic Actions

So far: *analysis* of the input

```
Expr = Term { "+" Term }.
```

the parser checks if the input is syntactically correct.

Now: *translation* of the input (semantic processing)

e.g.: we want to count the terms in the expression

```
Expr =
   Term          (. int n = 1; .)
   { "+" Term    (. n++; .)
   }             (. Console.WriteLine(n); .)
   .
```

semantic actions

- arbitrary Java statements between (. and .)
- are executed by the parser at the position where they occur in the grammar

"translation" here:

1+2+3	⇩	3
47+1	⇩	2
909	⇩	1

Attributes

Syntax symbols can return values (sort of output parameters)

Term <✸int val> *Term* returns its numeric value as an output attribute

Attributes are useful in the translation process

e.g.: we want to compute the value of a number

```
Expr                        (. int sum, val; .)
=  Term<✸sum>
   { "+" Term<✸val>         (. sum += val; .)
   }                        (. Console.WriteLine(sum); .)
   .
```

"translation" here:

1+2+3	⏻	6
47+1	⏻	48
909	⏻	909

Input Attributes

Nonterminal symbols can have also input attributes
(parameters that are passed from the "calling" production)

```
Expr<❄bool printHex>
```

printHex: print the result of the addition hexadecimal (otherwise decimal)

Example

```
Expr<❄bool printHex>      (. int sum, val; .)
=  Term<✸sum>
   { "+" Term<✸val>       (. sum += val; .)
   }.                     (.      if (printHex) Console.WriteLine("{0:X}", sum)
                                  else Console.WriteLine("{0:D}", sum);
                          .)
```

Attribute Grammars

Notation for describing translation processes

consist of three parts

1. **Productions in EBNF**

```
Expr = Term { "+" Term }.
```

2. **Attributes** (parameters of syntax symbols)

```
Term<✸int val>
Expr<❋bool printHex>
```

output attributes (*synthesized*): yield the translation result
input attributes (*inherited*): provide context from the caller

3. **Semantic actions**

```
(.   ... arbitrary Java statements ...   .)
```

Example

ATG for processing declarations

```
VarDecl                          (. Struct type; .)
=  Type <✸type>
   IdentList <✵type>
   ";" .
```

```
IdentLIst<✵Struct type>
=  ident                         (. Tab.insert(token.str, type); .)
   { "," ident                   (. Tab.insert(token.str, type); .)
   } .
```

This is translated to parsing methods as follows

```
static void VarDecl () {
   Struct type;
   Type(out type);
   IdentList(type);
   Check(Token.SEMICOLON);
}
```

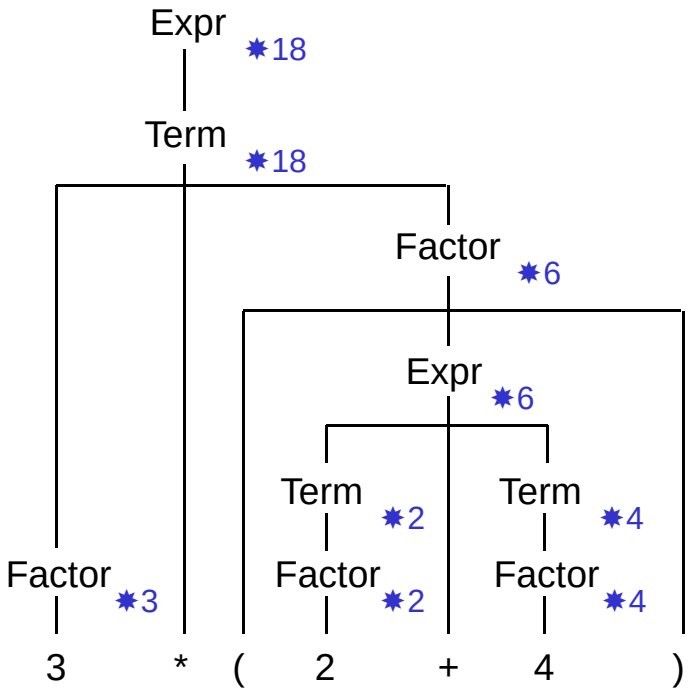
```
static void IdentList (Struct type) {
   Check(Token.IDENT);
   Tab.Insert(token.str, type);
   while (la == Token.COMMA) {
      Scan();
      Check(Token.IDENT);
      Tab.Insert(token.str, type);
   }
}
```

ATGs are shorter and more readable than parsing methods

Example: Processing of Constant Expressions

input: 3 * (2 + 4)
desired result: 18

```
Expr<✸int val>                (. int val1; .)
=  Term <✸val>
   { "+" Term <✸val1>         (. val += val1; .)
   |  "-" Term <✸val1>        (. val -= val1; .)
   }.

Term <✸int val>               (. int val1; .)
=  Factor <✸val>
   { "*" Factor <✸val1>       (. val *= val1; .)
   |  "/" Factor <✸val1>      (. val /= val1; .)
   }

Factor <✸int val>             (. int val1; .)
=  number                     (. val = t.val; .)
|  "(" Expr<✸val> ")"
```



Transforming an ATG into a Parser

Production

```
Expr<✸int val>              (. int val1; .)
=  Term<✸val>
   { "+" Term<✸val1>        (. val += val1; .)
   |  "-" Term<✸val1>       (. val -= val1; .)
   }.
```

Parsing method

```
static void Expr (out int val) {
   int val1;
   Term(out val);
   for (;;) {
      if (la == Token.PLUS) {
         Scan();
         val1 = Term(out val1);
         val += val1;
      } else if (la == Token.MINUS) {
         Scan();
         Term(out val1);
         val -= val1;
      } else break;
   }
}
```

input attribute ⇩ parameter
output atribute ⇩ *out* parameter
semantic actions ⇩ embedded Java code

Terminal symbols have no input attributes.

In our form of ATGs they also have no output attributes but their value is computed from *token.str or token.val*.

Example: Sales Statistics

ATGs can also be used in areas other than compiler constructions

Example: given a file with sales numbers

```
File    = { Article }.
Article = Code { Amount } "END"
Code    = number.
Amount  = number.
```

Whenever the input is syntacticlly structured
ATGs are a good notation to describe its processing

Input for example:

```
3451    2  5  3  7  END
3452    4  8  1 END
3453    1  1 END
...
```

Desired output:

```
3451    17
3452    13
3453    2
...
```

ATG for the Sales Statistics

```
File                                    (. int code, amount; .)
= { Article<✸code, ✸amount>             (. Write(code + " " + amount); .)
  }.

Article<✸int code, ✸int amount>
= Value<✸code>
  {                                     (. int x; .)
    Value<✸x>                           (. amount += x; .)
  }
  "END".

Value<✸int x>
= number                                (. x = token.val; .)
  .
```

Parsercode

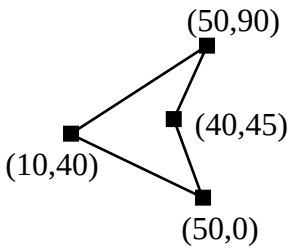
```
static void File () {
   int code, amount;
   while (la == number) {
      Article(out code, out number);
      Write(code + " " + amount);
   }
}
```

```
static void Article (out int code, out int amount) {
   Value(out code);
   while (la == number) {
      int x; Value(out x); amount += x;
   }
   Check(end);
}
```

```
static void Value (out int x) { Check(number); x = token.val; }
```

terminal symbols
number, end, eof

Example: Image Description Language



described by:

```
POLY
   (10,40)
   (50,90)
   (40,45)
   (50,0)
END
```

input syntax:

```
Polygon = "POLY" Point {Point} "END".
Point = "(" number "," number ")".
```

We want a program that reads the input and draws the polygon

```
Polygon                 (. Pt p, q; .)
=  "POLY"
   Point<✸p>            (. Turtle.start(p); .)
   { "," Point<✸q>      (. Turtle.move(q); .)
   }
   "END"                (. Turtle.move(p); .)
   .

Point<✸p>               (. Pt p; int x, y; .)
=  "(" number           (. x = t.val; .)
   "," number           (. y = t.val; .)
   ")"                  (. p = new Pt(x, y); .)
   .
```

We use "Turtle Graphics" for drawing

Turtle.start(p);	sets the turtle (pen) to point *p*
Turtle.move(q);	moves the turtle to *q* drawing a line

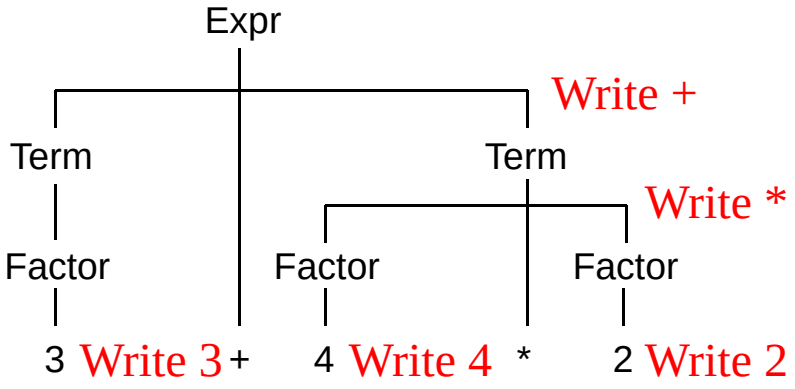
Example: Transform Infix to Postfix Expressions

Arithmetic expressions in infix notation are to be transformed to postfix notation

3 + 4 * 2 ⇒ 3 4 2 * +

(3 + 4) * 2 ⇒ 3 4 + 2 *

```
Expr =
   Term
   { "+" Term      (. Write("+"); .)
   |  "-" Term     (. Write("-"); .)
   }

Term =
   Factor
   { "*" Factor    (. Write("*"); .)
   |  "/" Factor   (. Write("/"); .)
   }.

Factor =
   number          (. Write(token.val); .)
|  "(" Expr ")".
```



5. Symbol Table

Responsibilities of the Symbol Table

1. It stores all declared names and their attributes

- type
- value (for constants)
- address (for local variables and method arguments)
- parameters (for methods)
- ...

2. It is used to retrieve the attributes of a name

- Mapping: name ↦ (type, value, address, ...)

Contents of the symbol table

- *Symbol* nodes: information about declared names
- *Structure* nodes: information about type structures

=> most suitably implemented as a dynamic data structure

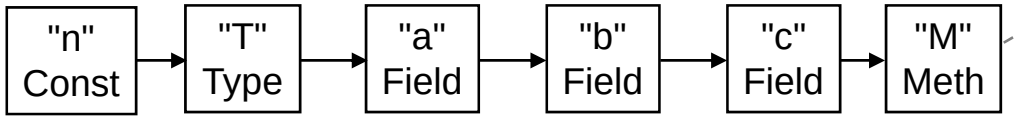
- linear list
- binary tree
- hash table

Symbol Table as a Linear List

Given the following declarations

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

we get the following linear list



for every declared name there is a Symbol node

\+ simple
\+ declaration order is retained (important if addresses are assigned only later)

\- slow if there are many declarations

Basic interface

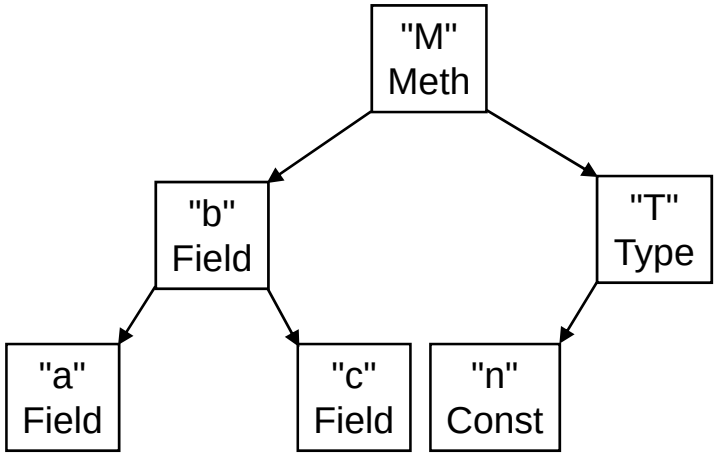
```
public class Tab {
   public static Symbol Insert (Symbol.Kinds kind, string name, ...);
   public static Symbol Find (string name);
}
```

Symbol Table as a Binary Tree

Declarations

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

Resulting binary tree



+ fast

- can degenerate unless it is balanced
- larger memory consumption
- declaration order is lost

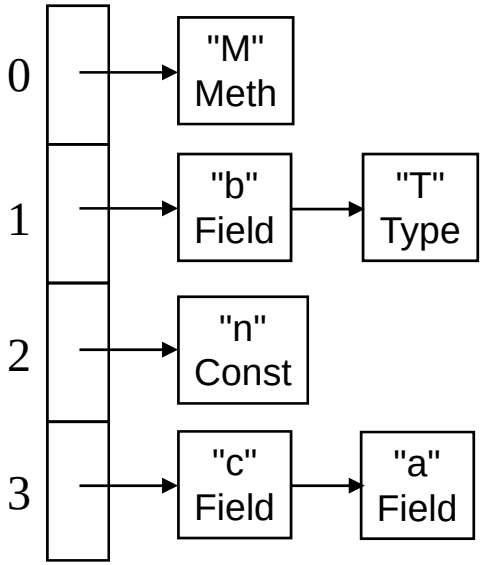
Only useful if there are many declarations

Symbol Table as a Hashtable

Declarations

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

Resulting hashtable



\+ fast

\- more complicated than a linear list
\- declaration order is lost

For our purposes a linear list is sufficient

- Every scope is a list of its own anyway
- A scope has hardly more than 10 names

5. Symbol Table

Symbol Nodes

Every declared name is stored in a Symbol node

Kinds of symbols in Z#

- constants
- global variables
- fields
- method arguments
- local variables
- types
- methods
- program

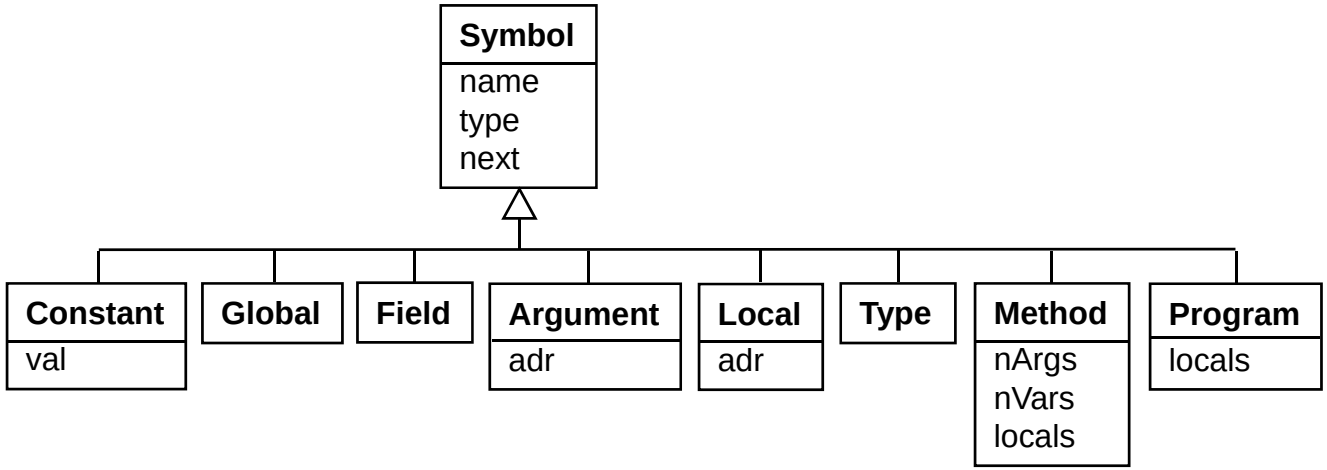
```
public enum Kinds {
   Const,
   Global,
   Field,
   Arg,
   Local,
   Type,
   Meth,
   Prog
}
```

What information is needed about objects?

- for all symbols — name, type structure, symbol kind, pointer to the next symbol
- for constants — value
- for method arguments — address (= order of declaration)
- for local variables — address (= order of declaration)
- for methods — number of arguments and local variables, local symbols (args + local vars)
- for program — global symbols (= local to the program)
- for global vars, fields, types — ---

Possible Object-oriented Architecture

Possible class hierarchy of objects



However, this is too complicated because it would require too many type casts

```
Symbol sym = Tab.Find("x");
if (sym is Argument) ((Argument) sym).adr = ...;
else if (sym is Method) ((Method) sym).nArgs = ...;
...
```

Therefore we choose a "flat implementation": all information is stored in a single class. This is ok because

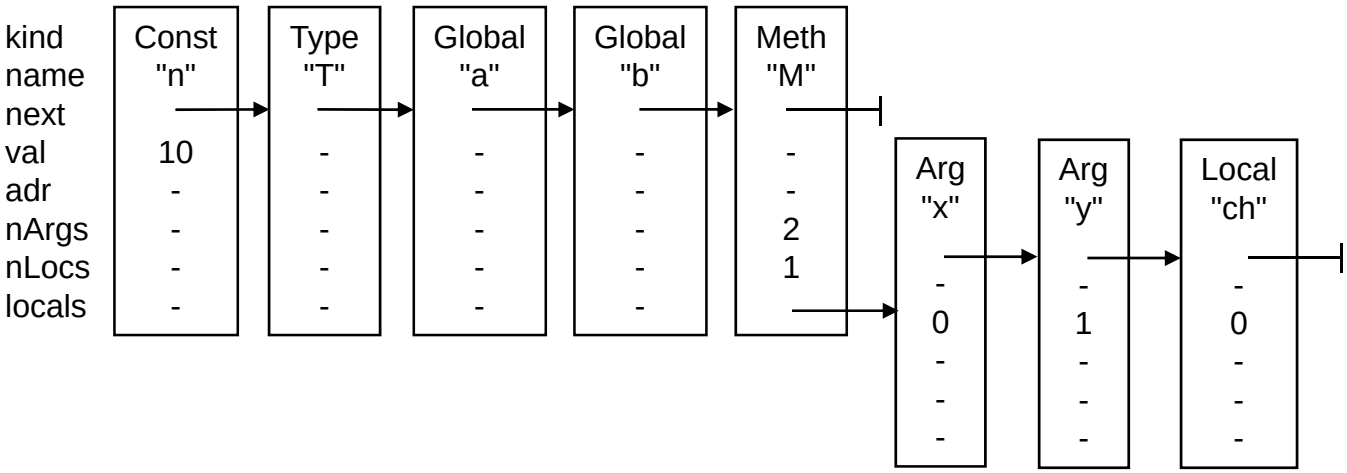
- extensibility is not required: we never need to add new object variants
- we do not need dynamically bound method calls

Class Symbol

```
class Symbol {
   public enum Kinds { Const, Global, Field, Arg, Local, Type, Meth, Prog }
   Kinds    kind;
   string   name;
   Struct   type;
   Symbol   next;
   int      val;      // Const: value
   int      adr;      // Arg, Local: address
   int      nArgs;    // Meth: number of arguments
   int      nLocs;    // Meth: number of local variables
   Symbol   locals;   // Meth: parameters & local variables; Prog: symbol table of program
}
```

Example

```
const int n = 10;
class T { ... }
int a, b;
void M (int x, int y)
   char ch;
{ ... }
```



Entering Names into the Symbol Table

The following method is called whenever a name is declared

```
Symbol sym = Tab.Insert(kind, name, type);
```

- creates a new object node with *kind*, *name*, *type*
- checks if *name* is already declared (if so => error message)
- assigns successive addresses to variables and fields
- enters the declaration level for variables (0 = global, 1 = local)
- appends the new node to the end of the symbol table list
- returns the new node to the caller

Example for calling *Insert()*

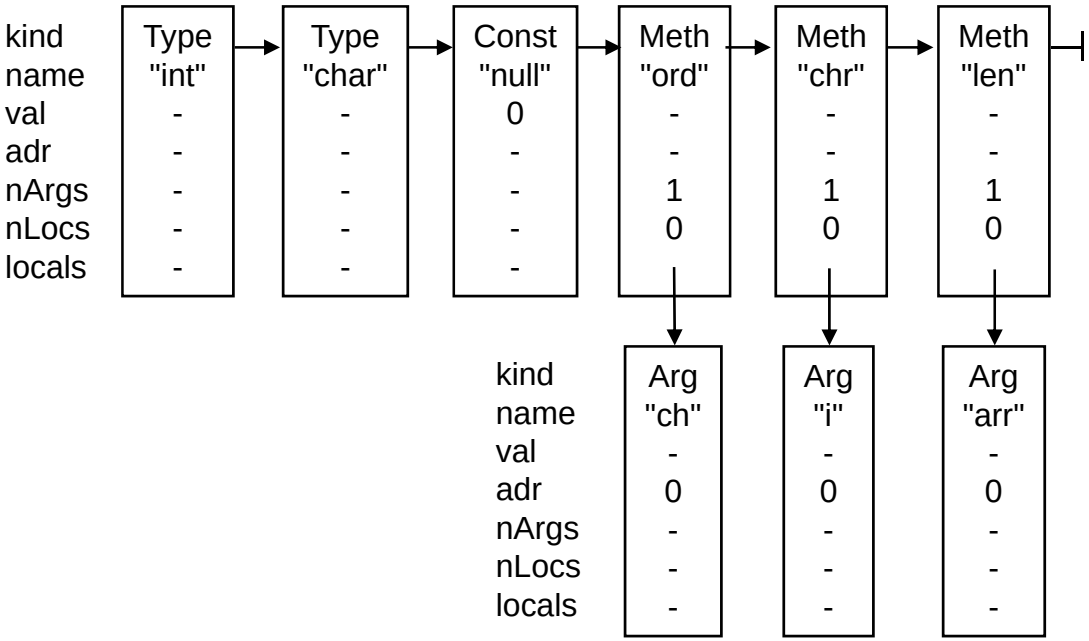
```
VarDecl<✵Symbol.Kinds kind>
=  Type<✸type>
   ident                    (. Tab.insert(Obj.Var, name, type); .)
   { ";" ident              (. Tab.insert(Obj.Var, name, type); .)
   }.
```

Predeclared Names

Which names are predeclared in Z#?

- Standard types: int, char
- Standard constants: null
- Standard methods: ord(ch), chr(i), len(arr)

Predeclared names are also stored in the symbol table ("Universe")



Special Names as Keywords

***int* and *char* could also be implemented as keywords.**

requires a special treatment in the grammar

```
Type<✸Struct type>
=  ident                (. Symbol sym = Tab.Find(token.str); type = sym.type; .)
|  "int"                (. type = Tab.intType; .)
|  "char"               (. type = Tab.charType; .)
.
```

It is simpler to have them predeclared in the symbol table.

```
Type<✸Struct type>
=  ident                (. Symbol sym = Tab.Find(token.str); type = sym.type; .)
```

\+ uniform treatment of predeclared and user-declared names

\- one can redeclare "int" as a user type

5. Symbol Table

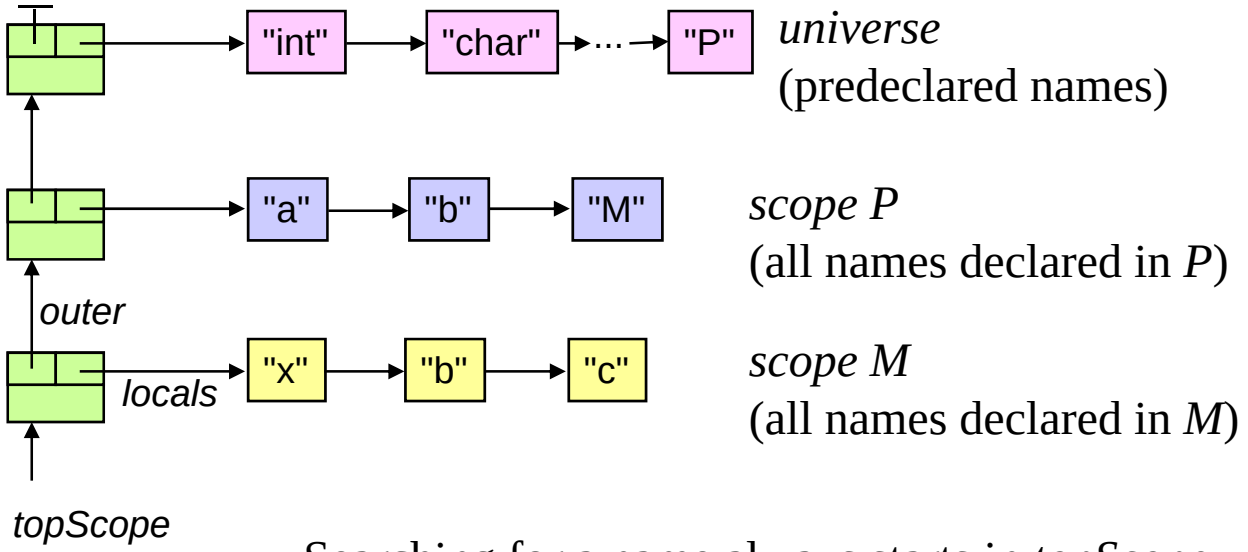
Scope = Range in which a Name is Valid

There are separate scopes (object lists) for

- the "universe" contains the predeclared names (and the program symbol)
- the program contains global names (= constants, global variables, classes, methods)
- every method contains local names (= argument and local variables)
- every class contains fields

Example

```
class P
   int a, b;
{
   void M (int x)
      int b, c;
   {
      ...
   }
   ...
}
```



- Searching for a name always starts in *topScope*
- If not found, the search continues in the next outer sc
- Example: search *b*, *a* and *int*

Scope Nodes


```
class Scope {
   Scope   outer;   // to the next outer scope
   Symbol  locals;  // to the symbols in this scope
   int     nArgs;   // number of arguments in this scope (for address allocation)
   int     nLocs;   // number of local variables in this scope (for address allocation)
}
```

Method for opening a scope

```
static void OpenScope () { // in class Tab
   Scope s = new Scope();
   s.nArgs = 0; s.nLocs = 0;
   s.outer = topScope;
   topScope = s;
}
```

- called at the beginning of a method or class
- links the new scope with the existing ones
- new scope becomes *topScope*
- *Tab.Insert()* always creates symbols in *topScope*

Method for closing a scope

```
static void CloseScope () { // in class Tab
   topScope = topScope.outer;
}
```

- called at the end of a method or class
- next outer scope becomes *topScope*

Entering Names in Scope

Names are always entered in *topScope*

```
class Tab {
   Scope topScope;  // Zeiger auf aktuellen Scope
   ...
   static Symbol Insert (Symbol.Kinds kind, string name, Struct type) {
      //--- create symbol node
      Symbol sym = new Symbol(name, kind, type);

      if (kind == Symbol.Kinds.Arg) sym.adr = topScope.nArgs++;
      else if (kind == Symbol.Kinds.Local) sym.adr = topScope.nLocs++;

      //--- insert symbol node
      Symbol cur = topScope.locals, last = null;
      while (cur != null) {
         if (cur.name == name) Error(name + " declared twice");
         last = cur; cur = cur.next;
      }
      if (last == null) topScope.locals = sym;
      else last.next = sym;
      return sym;
   }
   ...
}
```

Opening and Closing a Scope


```
MethodDecl                (. Struct type; .)
=  Type<✸type>
   ident                  (.  curMethod = Tab.insert(Symbol.Kinds.Meth, token.str, type);
                              Tab.OpenScope();
                           .)
   ...
   "{"
   ...
   "}"                    (.  curMethod.nArgs = topScope.nArgs;
                              curMethod.nLocs = topScope.nLocs;
                              curMethod.locals = Tab.topScope.locals;
                              Tab.CloseScope();
                           .)
   .
```

global variable

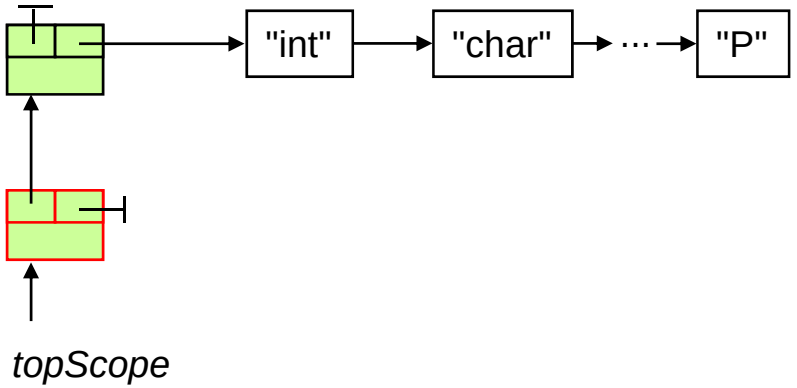
Note

- The method name is entered in the method's enclosing scope
- Before a scope is closed its local objects are assigned to *m.locals*
- Scopes are also opened and closed for classes

Example

class P

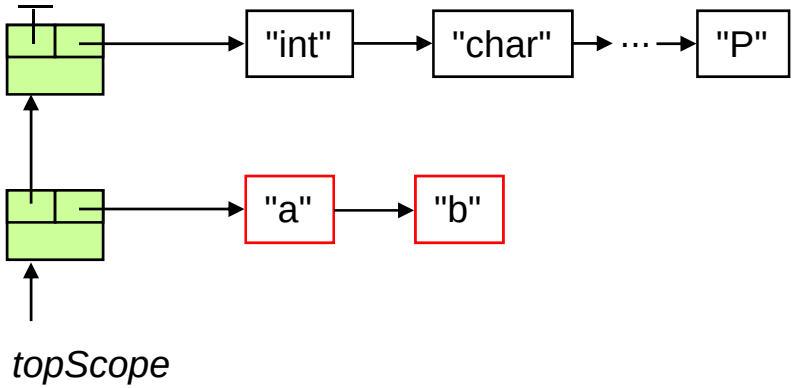
Tab.OpenScope();



Example

```
class P
    int a, b;
{
```

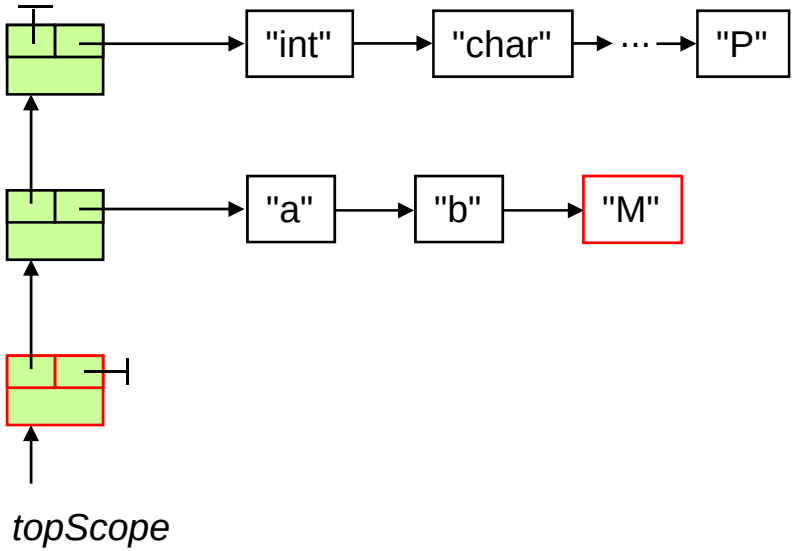
```
Tab.Insert(..., "a", ...);
Tab.Insert(..., "b", ...);
```



Example

```
class P
    int a, b;
{
    void M ()
```

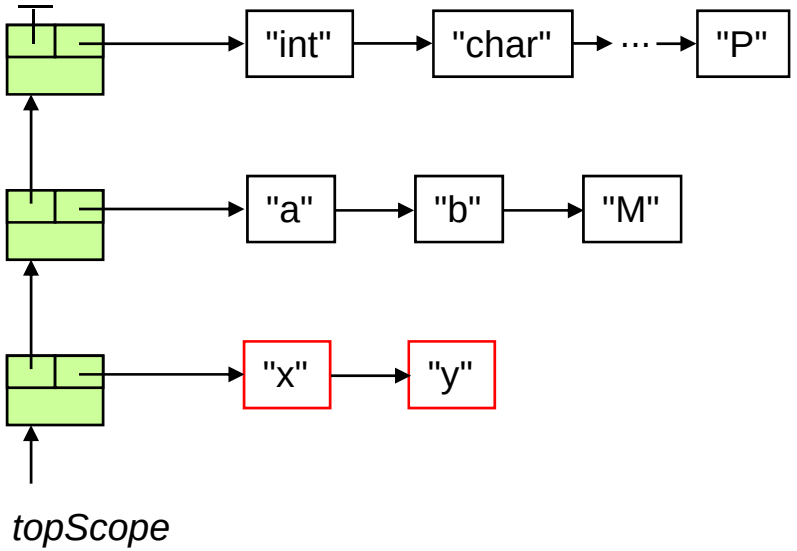
```
Tab.Insert(..., "M", ...);
Tab.OpenScope();
```



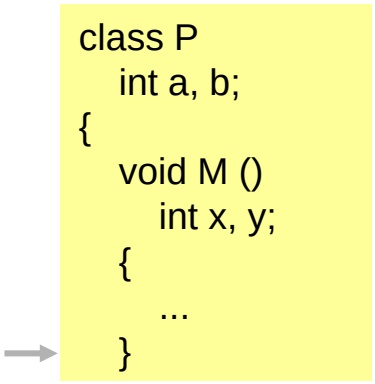
Example

```
class P
    int a, b;
{
    void M ()
        int x, y;
```

```
Tab.Insert(..., "x", ...);
Tab.Insert(..., "y", ...);
```

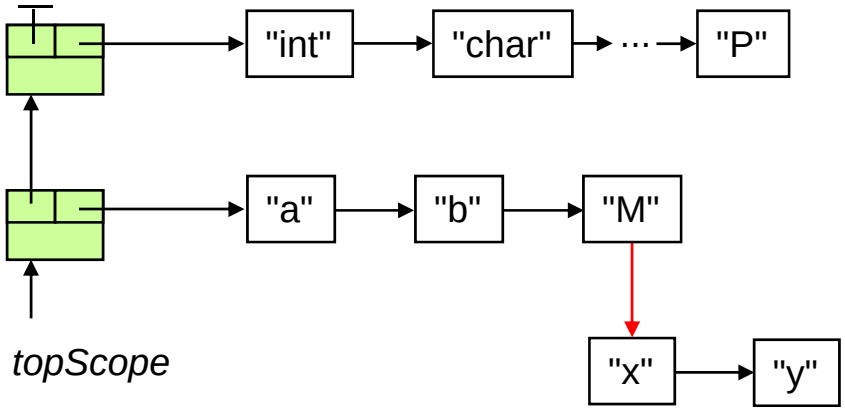


Example



```
class P
    int a, b;
{
    void M ()
        int x, y;
    {
        ...
    }
```

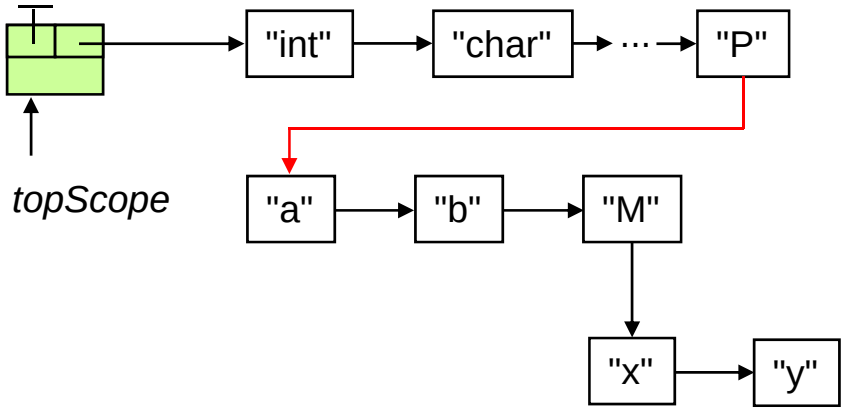
```
meth.locals =
    Tab.topScope.locals;
Tab.CloseScope();
```



Example

```
class P
   int a, b;
{
   void M ()
      int x, y;
   {
      ...
   }
   ...
}
```

```
prog.locals =
   Tab.topScope.locals;
Tab.CloseScope();
```



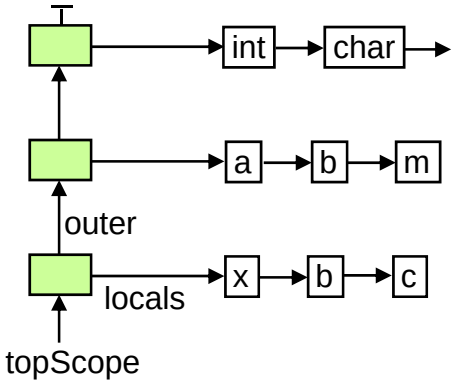
Searching Names in the Symbol Table

The following method is called whenever a name is used

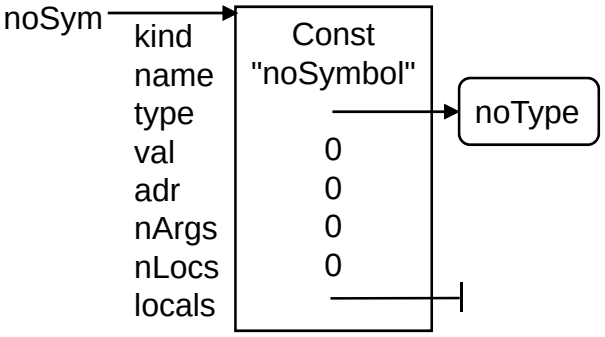
```
Symbol sym = Tab.Find(name);
```

- Lookup starts in *topScope*
- If not found, the lookup is continued in the next outer scope

```
static Symbol Find (string name) {
   for (Scope s = topScope; s != null; s = s.outer)
      for (Symbol sym = s.locals; sym != null; sym = sym.next)
         if (sym.name == name) return sym;
   Parser.Error(name + " is undeclared");
   return noSym;
}
```



If a name is not found the method returns *noSym*



- predeclared dummy symbol
- better than *null*, because it avoids aftereffects (exceptions)

5. Symbol Table

Types

Every object has a type with the following properties

- size (in Z# determined by metadata)
- structure (fields for classes, element type for arrays, ...)

Kinds of types in Z#?

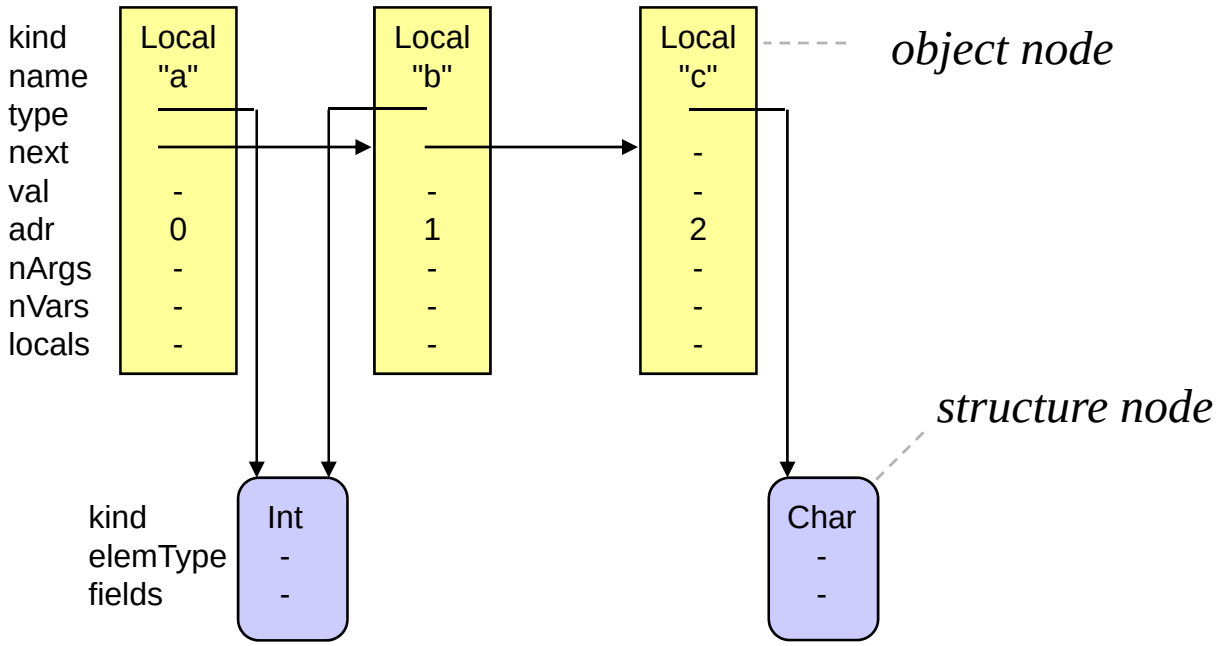
- primitive types (int, char)
- arrays
- classes

Types are represented by structure nodes

```
class Struct {
   public enum Kinds { None, Int, Char, Arr, Class }
   Kinds   kind;
   Struct  elemType;  // Arr: element type
   Symbol fields;     // Class: list of fields
}
```

Structure Nodes for Primitive Types

```
int a, b;
char c;
```

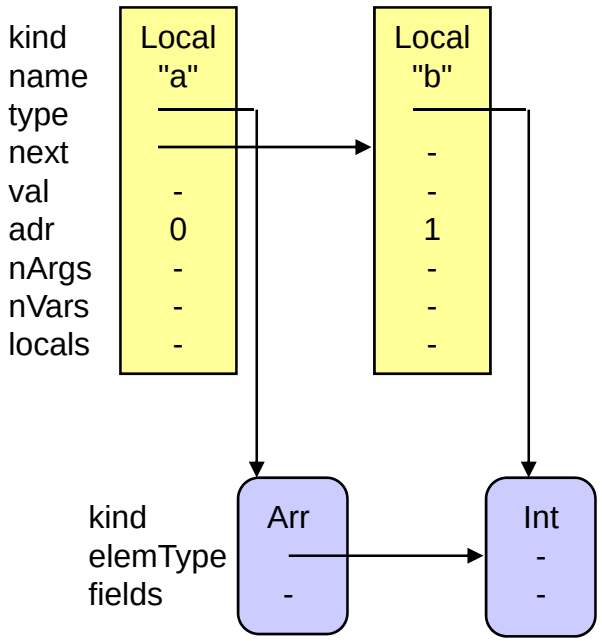


There is just one structure node for *int* in the whole symbol table. All symbols of type *int* reference this one.

The same is true for structure nodes of type *char*.

Structure Nodes for Arrays

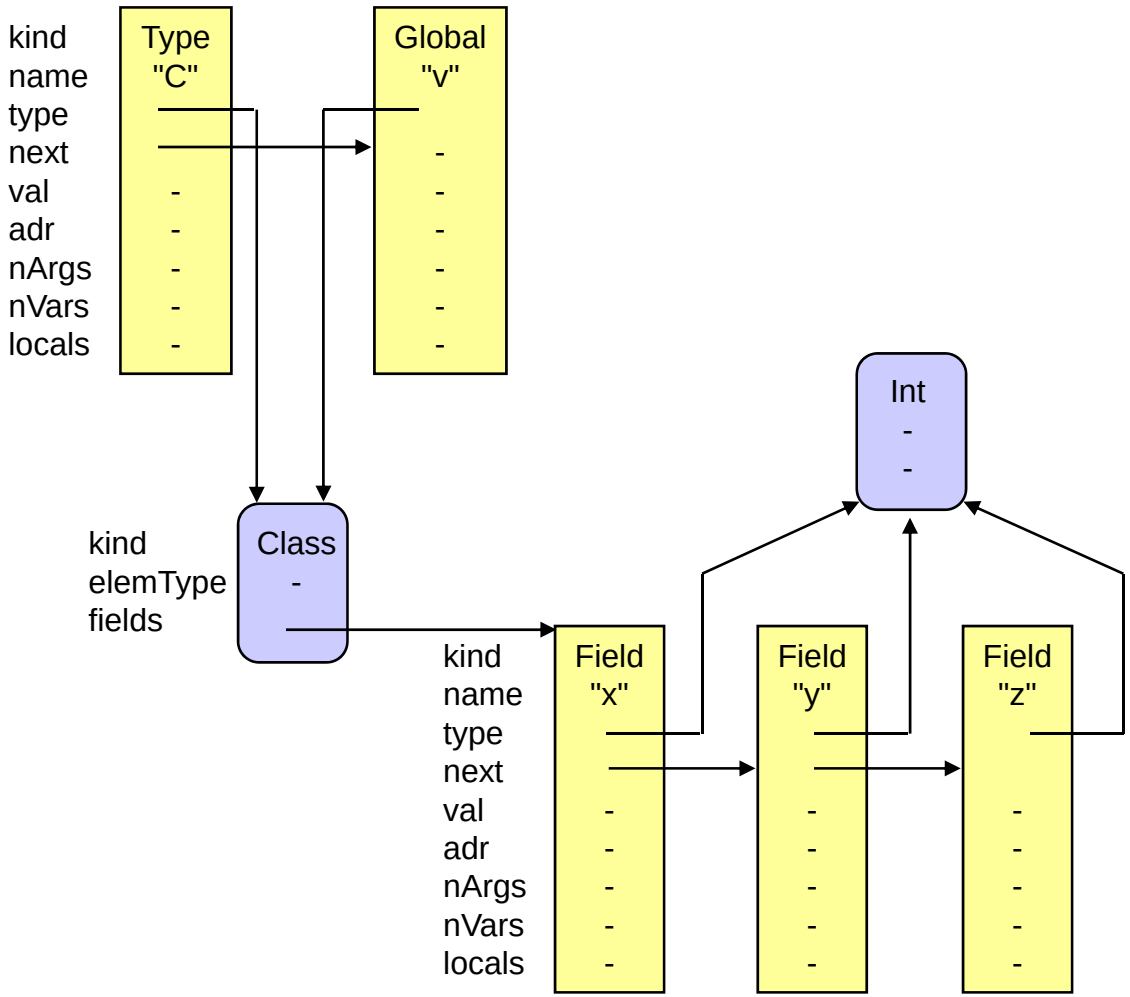
```
int[] a;
int b;
```



The length of an array is statically unknown.
It is stored in the array at run time.

Structure Nodes for Classes

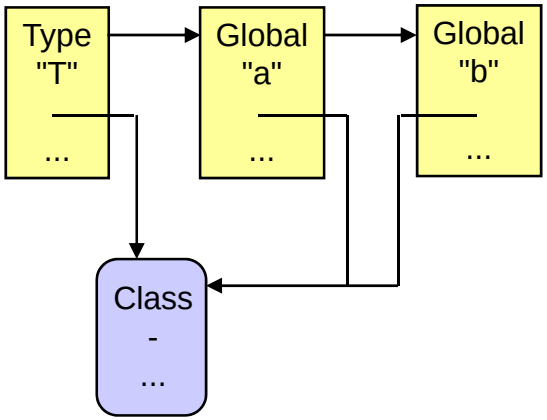
```
class C {
    int x;
    int y;
    int z;
}
C v;
```



Type Compatibility: *Name Equivalence*

Two types are equal if they are represented by the same type node
(i.e. if they are denoted by the same type name)

```
class T { ... }
T a;
T b;
```



The types of *a* and *b* are the same

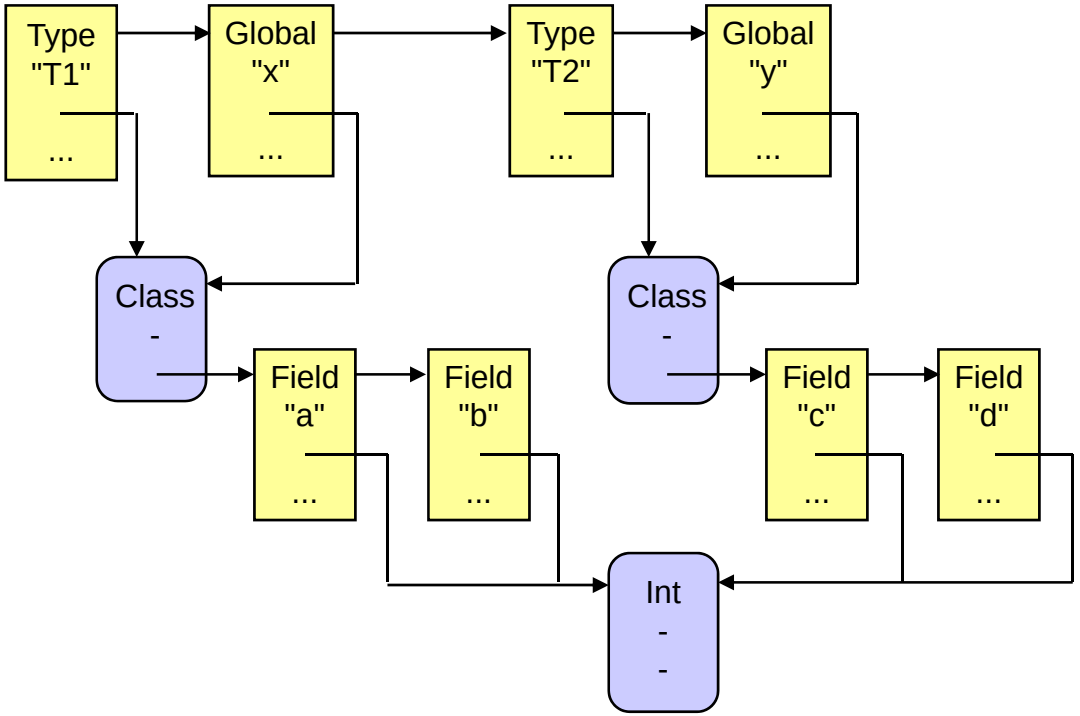
Name equivalence is used in Java, C/C++/C#, Pascal, ..., Z#

Exception
In C# (and Z#) two array types are the same if they have the same element types!

Type Compatibility: Structural Equivalence

Two types are the same if they have the same structure
(i.e. the same fields of the same types, the same element type, ...)

```
class T1 { int a, b; }
class T2 { int c, d; }
T1 x;
T2 y;
```



The types of *x* and *y* are equal (but not in Z#!)

Structural equivalence is used in Modula-3 but not in Z# and most other languages!

Methods for Checking Type Compatibility


```
class Struct {
   ...
   // checks, if two types are compatible (e.g. in comparisons)
   public bool CompatibleWith (Struct other) {
      return this.Equals(other) ||
             this == Tab.nullType && other.IsRefType() ||
             other == Tab.nullType && this.isRefType();
   }

   // checks, if this can be assigned to dest
   public bool AssignableTo (Struct dest) {
      return this.Equals(dest) ||
             this == Tab.nullType && dest.IsRefType() ||
             kind == Kinds.Arr && dest.kind == Kinds.Arr && dest.elemType == Tab.objType;
   }

   // checks, if two types are equal (structural equivalence for array, name equivalence otherwise)
   public bool Equals (Struct other) {
      if (kind == Kinds.Arr)
         return other.kind == Kinds.Arr && elemType.Equals(other.elemType);
      return other == this;
   }

   public bool IsRefType() { return kind == Kinds.Class || kind = Kinds.Arr; }
}
```

necessary for standard function len(arr)

Solving LL(1) Conflicts with the Symbol Table

Method syntax in Z#

```
void Foo ()
    int a;
{
    a = 0; ...
}
```

Actually we are used to write it like this

```
void Foo () {
    int a;
    a = 0; ...
}
```

But this would result in an LL(1) conflict

First(*VarDecl*) ⇘ First(*Statement*) = {*ident*}

```
Block       =  "{" { VarDecl | Statement } "}".
VarDecl     =  Type ident { "," ident }.
Type        =  ident [ "[" "]" ].
Statement   =  Designator "=" Expr ";"
            |  ... .
Designator  =  ident { "." ident | "[" Expr "]" }.
```

Solving the Conflict With Semantic Information

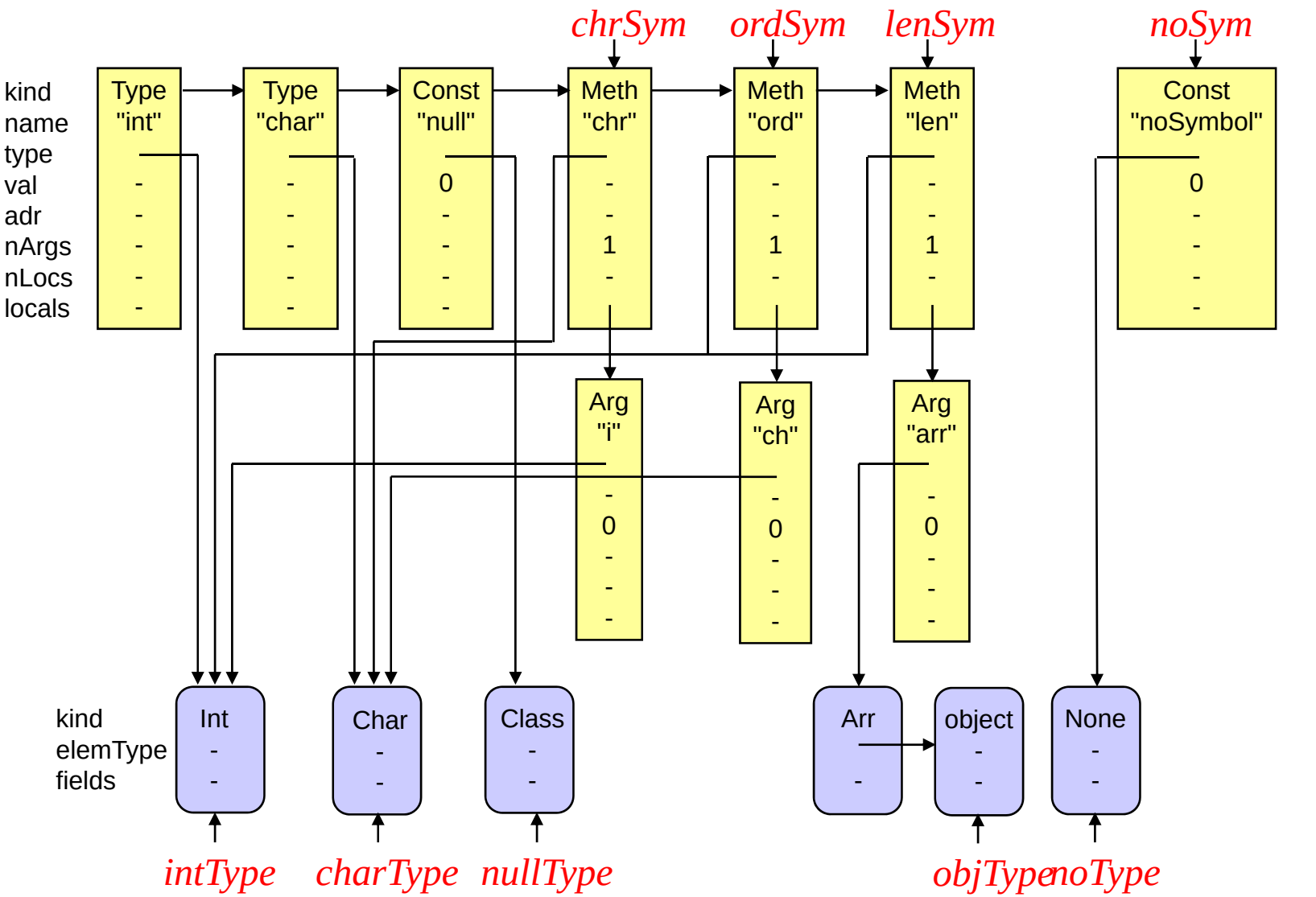

```
static void Block () {
   Check(Token.LBRACE);
   for (;;) {
      if (NextTokenIsType()) VarDecl();
      else if (la ∉ First(Statement))
Statement();
      else if (la ∉ {rbrace, eof}) break;
      else {
         Error("..."); ... recover ...
      }
   }
   Check(Token.RBRACE);
}
```

```
static bool NextTokenIsType() {
   if (la != ident) return false;
   Symbol sym = Tab.Find(laToken.str);
   return sym.kind == Symbol.Kinds.Type;
}
```

Block = "{" { VarDecl | Statement } "}".

5. Symbol Table

Structure of the "universe"



Interface of the Symbol Table


```
class Tab {
   static Scope    topScope;    // current top scope

   static Struct   intType;     // predefined types
   static Struct   charType;
   static Struct   nullType;
   static Struct   noType;

   static Symbol   chrSym;      // predefined symbols
   static Symbol   ordSym;
   static Symbol   lenSym;
   static Symbol   noSym;

   static Symbol   Insert (Symbol.Kinds kind, string name, Struct type) {...}
   static Symbol   Find (string name) {...}
   static void     OpenScope () {...}
   static void     CloseScope () {...}

   static void     Init () {...}    // builds the universe and initializes Tab
}
```

6. Code Generation

Responsibilities of the Code Generation

Generation of machine instructions

- selecting the right instructions
- selecting the right addressing modes

Translation of control structures (if, while, ...) into jumps

Allocation of stack frames for local variables

Maybe some optimizations

Output of the object file

Common Strategy

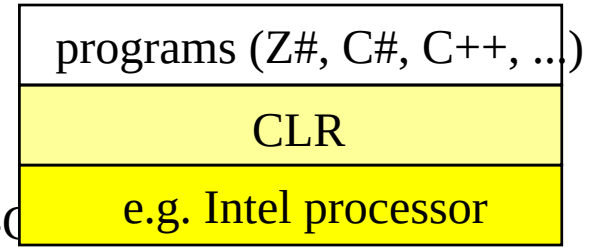
1. **Study the target machine**
   registers, data formats, addressing modes, instructions, instruction formats, ...
2. **Design the run-time data structures**
   layout of stack frames, layout of the global data area, layout of heap objects, layout of the constant pool, ...
3. **Implement the code buffer**
   instruction encoding, instruction patching, ...
4. **Implement register allocation**
   irrelevant for Z#, because we have a stack machine
5. **Implement code generation routines** (in the following order)
   - load values and addresses into registers (or onto the stack)
   - process designators (x.y, a[i], ...)
   - translate expressions
   - manage labels and jumps
   - translate statements
   - translate methods and parameter passing

6. Code Generation

Architecture of the CLR

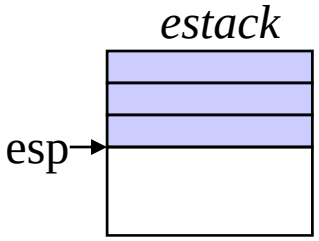
What is a virtual machine (VM)?

- a CPU implemented in Software
- Commands will be interpreted / JIT-compiled
- other examples: Java-VM, Smalltalk-VM, Pascal P-C



The CLR is a stack machine

- no registers
- instead it has an *expression stack* (onto which values are loaded)



max. size is stored in metadata of each method

esp ... expression stack pointer

The CLR executes JIT-compiled bytecode

- each method is compiled right before the first execution (= just-in-time)
- operands are addressed symbolically in IL (informationen is stored in the metadata)

How a Stack Machine Works

Example

statement **i = i + j * 5;**

assume the following values of *i* and *j*

locals

0	3	i
1	4	j

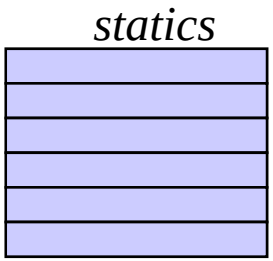
Simulation

instructions	*stack*	
ldloc.0	3	load variable from address 0 (i.e. *i*)
lldloc.1	3 4	load variable from address 1 (i.e. *j*)
ldc.i4.5	3 4 5	load constant 5
mul	3 20	multiply the two topmost stack elements
add	23	add the two topmost stack elements
stloc.0		store the topmost stack element to address 0

At the end of every statement the expression stack is empty!

Data Areas of the CLR

Global variables



- are mapped to static fields of the program class in the CLR
- global variables live during the whole program (= while the program class is loaded)
- global variables are addressed by medadata tokens
  e.g. ldsfld T_{fld} loads the value of the field referenced by T_{fld} onto *estack*

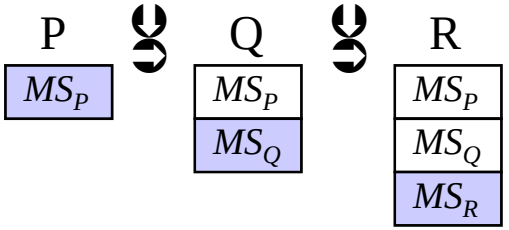
Metadata token are 4 Byte values that reference rows in metadata tables.

token type (1 Byte)	index into metadata table (3 Byte)

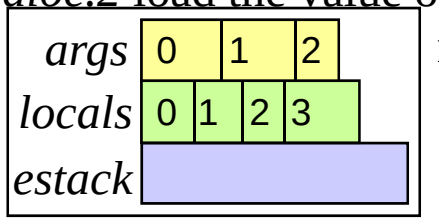
Data Areas of the CLR

Method state

- manages separate areas for
  - arguments (*args*)
  - local variables (*locals*)
  - expression stack (*estack*)
- each method call has its own method state (*MS*)
- method states managed in a stack and
  therefore also called *stack frames*.



- each parameter and each local variable occupy a slot of typedependent size.
- addresses are consecutive number reflecting the order of declaration
  e.g. *ldarg.0* loads the value of the first method argument onto *estack*
  ldloc.2 load the value of the third local variable onto *estack*

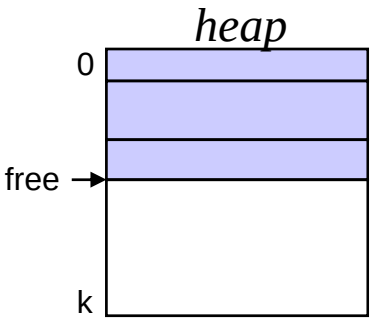


method state

Data Areas of the CLR

Heap

- contains class objects and array objects

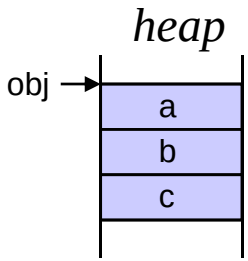


- new objects are allocated at the position *free* (and *free* is incremented); this is done by the CIL instructions *newobj* and *newarr*
- objects are deallocated by the garbage collector

Data Areas of the CLR

class objects

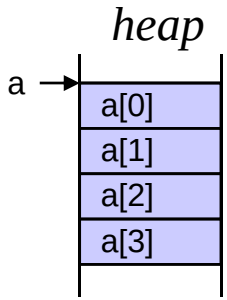
```
class X {
    int a, b;
    char c;
}
X obj = new X;
```



- addressed by field token relative to *obj*

array objects

```
int[] a = new int[4];
```



- addressed by index value relativ to *a*

- Z# supports only vectors, i.e. one dimensional arrays with index starting at 0
- only vectors can be handled by the special CIL instructions *newarr*, *ldlen*, *ldelem*, *stelem*

Code Buffer

- Z# has only static methods that belong to type of the main program class
- each method consists of a stream of CIL instructions and metadata, like *.entrypoint*, *.locals*, *.maxstack*, access attributes, ...

System.Reflection.Emit.ILGenerator manages the CIL stream.
We can access the IL stream of the currently compiled method via the static field

```
internal static ILGenerator il;
```

of class *Code*.

```
class ILGenerator {
    /* overloaded (see next slide) */
    virtual void Emit (OpCode op, ...);
    /* for method calls */
    void EmitCall (
        OpCode op,
        MethodInfo meth,
        Type[] varArgs);
    ...
}
```

Handles for instruction are defined in calss *OpCo*
Wir verwenden Abkürzungen in Klasse *Code*.

```
static readonly OpCode
    LDARG0 = OpCodes.Ldarg_0, ...
    LDARG  = OpCodes.Ldarg_S, ...
    LDC0   = OpCodes.Ldc_I4_0, ...
    ADD    = OpCodes.Add, ...
    BEQ    = OpCodes.Beq, ...
    ... ;
```

e.g.: Generating *ldloc.2*

```
Code.il.Emit(Code.LDLOC2);
```

Emit methods of ILGenerator

```
class ILGenerator {                          // use for the following IL instructions:
   void Emit (OpCode);                       // ldarg.n, ldloc.n, stloc.n, ldnull, ldc.i4.n, ld.i4.m1
                                             // add, sub, mul, div, rem, neg,
                                             // ldlen, ldelem... , stelem... , dup, pop, ret, throw
   void Emit (OpCode, byte);                 // ldarg.s, starg.s, ldloc.s, stloc.s
   void Emit (OpCode, int);                  // ldc.i4
   void Emit (OpCode, FieldInfo);            // ldsfld, stsfld, ldfld, stfld
   void Emit (OpCode, LocalBuilder);         // ldloc.s, stloc.s
   void Emit (OpCode, ConstructorInfo);      // newobj
   void Emit (OpCode, Type);                 // newarr
   void Emit (OpCode, Label);                // br, beq, bge, bgt, ble, blt, bne.un

   /* for method calls */
   void EmitCall (OpCode, MethodInfo, Type[]);  // call (set last argument (varargs) to null)
   ...
}
```

Metadata

describes the properties of the components of an assembly (types, fields, methods, ...).
Class *Code* has fields for managing important metadata items:

```
static AssemblyBuilder assembly; // the program assembly
static ModuleBuilder module;     // the program module
static TypeBuilder program;      // the main class
static TypeBuilder inner;        // the currently compiled inner class
```

The method *CreateMetadata* of class *Code* creates metadata objects from symbol nodes:

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Prog:
         AssemblyName aName = new AssemblyName(); aName.Name = sym.name;
         assembly = AppDomain.CurrentDomain.DefineDynamicAssembly(
                              aName, AssemblyBuilderAccess.Save);
         module = assembly.DefineDynamicModule(sym.name + "Module", sym.name + ".exe");
         program = module.DefineType(sym.name, TypeAttributes.Class | TypeAttributes.Public);
         inner = null;
         break;
      ...
}
```

Metadata: local variables

The method *DeclareLocal* of *ILGenerator* returns a System.Reflection.Emit.LocalBuilder that builds the metadata for local variables.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Local:
         il.DeclareLocal(sym.type.sysType);
         break;
      ...
}
```

- additional field in Struct nodes

  ```
  internal Type sysType;
  ```

  for creating the corresponding CLR types for a Symbol node.

Metadata: types

The method *DefineType* of *ModuleBuilder* returns a System.Reflection.Emit.TypeBuilder that builds the metadata for types.
Inner classes from Z# are mapped to outer types in the CLR.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      inner = module.DefineType(sym.name, TypeAttributes.Class | TypeAttributes.NotPublic);
      sym.type.sysType = inner;

      // define default contructor (simply calls base constructor)
      sym.ctor = inner.DefineConstructor(  MethodAttributes.Public, CallingConventions.Standard,
                                           new Type[0]);
      il = sym.ctor.GetILGenerator();
      il.Emit(LDARG0);
      il.Emit(CALL, typeof(object).GetConstructor(new Type[0]));
      il.Emit(RET);
      break;
      ...
}
```

no-arg constructor of System.Object

- additional field for Symbol nodes

```
internal ConstructorBuilder ctor;
```

for generating invocations to constructor when creating new objects (*newobj*).

Metadata: global variables & object fields

The CLR handles both as fields (static fields of program class; instance fields of inner class
The method *DefineField* of *TypeBuilder* returns a System.Reflection.Emit.FieldBuilder that generates the metadata for fields.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Global:
         if (sym.type != Tab.noType)
            sym.fld = program.DefineField(sym.name, sym.type.sysType, FieldAttributes.Assembly
                                                                    | FieldAttributes.Static);
         break;
      case Symbol.Kinds.Field:
         if (sym.type != Tab.noType)
            sym.fld = inner.DefineField(sym.name, sym.type.sysType, FieldAttributes.Assembly);
         break;
      ...
}
```

- additional fields for Symbol nodes

  ```
  internal FieldBuilder fld;
  ```

  for generating field accesses (*ldsfld*, *stsfld*, *ldfld*, *stfld*).

Metadata: methods & arguments

The method *DefineMethod* of *TypeBuilder* returns a System.Reflection.Emit.MethodBuilder that generates the metadata for methods and their arguments.

```
internal static void CreateMetadata (Symbol sym) {
   switch (sym.kind) {
      case Symbol.Kinds.Meth:
         Type[] args = new Type[sym.nArgs];
         Symbol arg;
         for (arg = sym.locals; arg != null && arg.kind == Symbol.Kinds.Arg; arg = arg.next)
            args[arg.adr] = arg.type.sysType;
         sym.meth = program.DefineMethod( sym.name,   MethodAttributes.Static
                                                    | MethodAttributes.Family,
                                          sym.type.sysType, args);
         il = sym.meth.GetILGenerator();
         if (sym.name == "Main") assembly.SetEntryPoint(sym.meth);
         break;
      ...
}
```

- additional field for Symbol nodes

  ```
  internal MethodBuilder meth;
  ```

  for generating method calls (*call*).

Instruction Set of the CLR

Common Intermediate Language (CIL) (here we need only a subset)

- very compact: most instructions are just 1 byte long
- mostly untyped; type indications refer to the result type

untyped	*typed*
ldloc.0	ldc.i4.3
starg.1	ldelem.i2
add	stelem.ref

Instruction format

very simple compared to Intel, PowerPC or SPARC

```
Code        = { Instruction }.
Instruction = opcode [ operand ].
```

opcode ... 1 or 2 byte
operand ... of primitive type or metadata to

Examples

0 operands	add	has 2 implicit operands on the stack
1 operand	ldc.i4.s 9	

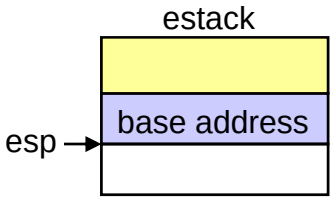
Instruction Set of the CLR

Addressing modes

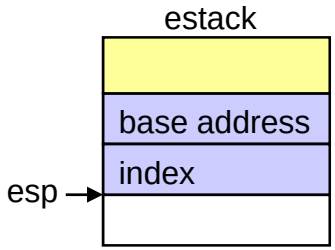
How can operands be accessed?

addressing mode	*example*	
• **Immediate**	ldc.i4 123	for constants
• **Arg**	ldarg.s 5	for method arguments
• **Local**	ldlocs.s 12	for local variable
• **Static**	ldsfld *fld*	for statische Felder (*fld* = metadata token)
• **Stack**	add	for loaded values on *estack*
• **Relative**	ldfld *fld*	for object fields (object reference is on *estack*)
• **Indexed**	ldelem.i4	for array elements (array reference & index are on *estack*)

Relative



Indexed



Instruction Set of the CLR

Loading and storing of method arguments

ldarg.s b	... ..., val	Load push(args[b]);
ldarg.*n*	... ..., val	Load (n = 0..3) push(args[n]);
starg.s b	..., val ...	Store args[b] = pop();

Loading and storing of local variables

ldloc.s b	... ..., val	Load push(locals[b]);
ldloc.*n*	... ..., val	Load (n = 0..3) push(locals[n]);
stloc.s b	..., val ...	Store locals[b] = pop();
stloc.*n*	..., val ...	Store (n = 0..3) locals[n] = pop();

Operand types

b ... unsigned byte
i ... signed integer
T ... metadata token

Instruction Set of the CLR

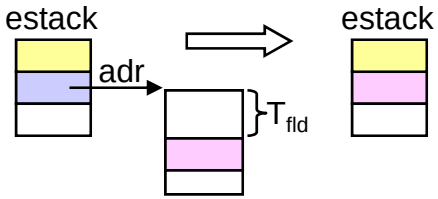
Loading and storing of global variables

ldsfld	T_{fld}	... ..., val	Load static variable push(statics[T_{fld}]);
stsfld	T_{fld}	..., val ...	Store static variable statics[T_{fld}] = pop();

Loading and storing of object field

ldfld	T_{fld}	..., obj
		..., val
			Load object field
			obj = pop();
			push(heap[obj+T_{fld}]);

stfld	T_{fld}	..., obj, val
		...
			Store object field
			val = pop(); obj = pop();
			heap[obj+T_{fld}] = val;

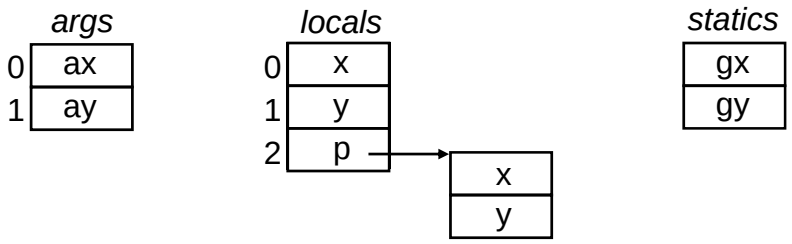


Instruction Set of the CLR

Loading of constants

ldc.i4 i	... ..., i	Load constant push(i);
ldc.i4.*n*	... ..., *n*	Load constant (n = 0..8) push(n);
ldc.i4.m1	... ..., -1	Load minus one push(-1);
ldnull	... ..., null	Load null push(null);

Example: loading and storing



	Code	*Bytes*	*estack*
ax = ay;	ldarg.1	1	ay
	starg.s 0	2	-
x = y;	ldloc.1	1	y
	stloc.0	1	-
gx = gy;	ldsfld T_{fld_gy}	5	gy
	stsfld T_{fld_gx}	5	-
p.x = p.y;	ldloc.2	1	p
	ldloc.2	1	p p
	ldfld T_{fld_y}	5	p p.y
	stfld T_{fld_x}	5	-

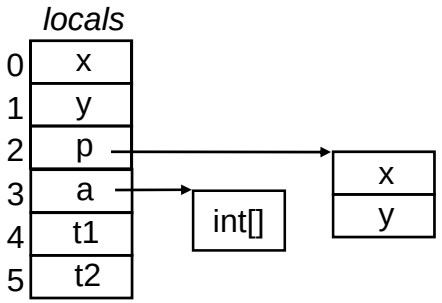
Instruction Set of the CLR

Arithmetics

add	..., val1, val2	Add push(pop() + pop());
sub	**...,** val1, val2	Subtract push(-pop() + pop());
mul	..., val1, val2	Multiply push(pop() * pop());
div	**...,** val1, val2	Divide x = pop(); push(pop() / x);
rem	..., val1, val2	Remainder x = pop(); push(pop() % x);
neg	**...,** val	Negate push(-pop());

Examples: arithmetics

	Code	*Bytes*	*Stack*
x + y * 3	ldloc.0	1	x
	ldloc.1	1	x y
	ldc.i4.3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3



x++;	ldloc.0	1	x
	ldc.i4.1	1	x 1
	add	1	x+1
	stloc.0	1	-

p.x++	ldloc.2	1	p
	dup	1	p p
	ldfld T_{px}	5	p p.x
	ldc.i4.1	1	p p.x 1
	add	1	p p.x+1
	stfld T_{px}	5	-

x--;	ldloc.0	1	x
	ldc.i4.m1	1	x -1
	add	1	x-1
	stloc.0	1	-

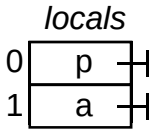
a[2]++	ldloc.3	1	a
	ldc.i4.2	1	a 2
	stloc.s 5	2	a
	stloc.s 4	2	-
	ldloc.s 4	2	a
	ldloc.s 5	2	a 2
	ldloc.s 4	2	a 2 a
	ldloc.s 5	2	a 2 a 2
	ldelem.i4	1	a 2 a[2]
	ldc.i4.1	1	a 2 a[2] 1
	add	1	a 2 a[2]+1
	stelem.i4	1	-

Instruction Set of the CLR

Object creation

newobjT_{ctor}	... [arg0, ..., argN] ..., obj	New object creates a new object of the type specified by the constructor token and then executes the constructor (arguments are on the stack)
newarrT_{eType}	..., n ..., arr	New array creates an array with space for n elements of the type specified by the type token

Examples: object creation



	Code	*Bytes*	*Stack*
Person p = new Person;	newobj $T_{P()}$	5	p
	stloc.0	1	-
int[] a = new int[5];	ldc.i4.5	1	5
	newarr T_{int}	5	a
	stloc.1	1	-

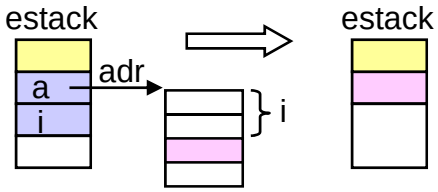
Instruction Set of the CLR

Array access

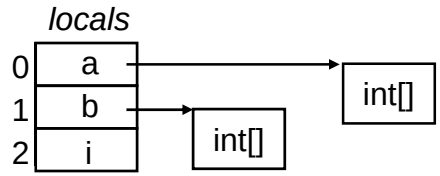
ldelem.u2
ldelem.i4
ldelem.ref
	..., adr, i
	..., val
		Load array element
		i = pop(); adr = pop();
		push(heap[adr+i]);
		result type on *estack*:
		char, int, object reference

stelem.i2
stelem.i4
stelem.ref
	...,adr, i,
	val
	...
		Store array element
		val=pop(); i=pop();
		adr=pop()/4+1;
		heap[adr+i] = val;
		type of element to be stored:

ldlen	..., adr ..., len	Get array length



Example: array access



	Code	*Bytes*	*Stack*
a[i] = b[i+1];	ldloc.0	1	a
	ldloc.2	1	a i
	ldloc.1	1	a i b
	ldloc.2	1	a i b i
	ldc.i4.1	1	a i b i 1
	add	1	a i b i+1
	ldelem.i4	1	a i b[i+1]
	stelem.i4	1	-

Instruction Set of the CLR

Stack manipulation

pop	..., val ...	Remove topmost stack element dummy = pop();
dup	..., val ..., val, val	Duplicate topmost stack element x = pop(); push(x); push(x);

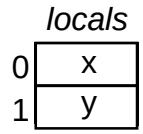
Jumps

br i	... ...	Branch unconditionally pc = pc + i

b<*cond*>i	..., x, y
	...
		Branch conditionally
		(<*cond*> = eq | ge | gt | le | lt | ne.un)
		y = pop(); x = pop();
		if (x cond y) pc = pc + i;

pc marks the current instruction;
i (jump distance) relative to the start of the next instruction

Example: jumps



	Code	*Bytes*	*Stack*
if (x > y) {	ldloc.0	1	x
...	ldloc.1	1	x y
}	ble ...	5	-
...			

Instruction Set of the CLR

Method call

call T_{meth}	... [arg0, ... argN] ... [retVal]	Call method pops the arguments off caller *estack* and puts them into *args* of the callee; pops return value off callee *estack*
ret	... ...	Return from method

Miscellaneous

throw	..., exc ...	Throw exception

Instruction Set of the CLR

I/O

The functionality of the Z# keywords *read* and *write* is provided by separate methods of the main program class. These use the class *System.Console* for In-/Output.

```
static char read ();
static int readi ();
static void write (char, int);
static void write (int, int);
```

These methods are added to the main program class by the compiler.

```
internal static void CreateMetadata (Symbol sym) { ...
   switch (sym.kind) { ...
      case Symbol.Kinds.Program: ...
         BuildReadChar(); BuildReadInt(); BuildWriteChar(); BuildWriteInt(); break;
   }
}
```

and are then accessible via static fields of class *Code*:

```
internal static MethodBuilder readChar, readInt, writeChar, writeInt;
```

In the parser we can generate a call to these methods like this:

```
Code.il.EmitCall(Code.CALL, Code.readChar, null);
```

Example

```
void Main ()
   int a, b, max, sum;
{                                  static void Main ()
   if (a > b)                        0:  ldloc.0
                                     1:  ldloc.1
                                     2:  ble 7 (=14)
      max = a;                       7:  ldloc.0
                                     8:  stloc.2
                                     9:  br 2 (=16)
   else max = b;                    14:  ldloc.1
                                    15:  stloc.2
   while (a > 0) {                  16:  ldloc.0
                                    17:  ldc.i4.0
                                    18:  ble 15 (=38)
      sum = sum + a * b;            23:  ldloc.3
                                    24:  ldloc.0
                                    25:  ldloc.1
                                    26:  mul
                                    27:  add
                                    28:  stloc.3
      a--;                          29:  ldloc.0
                                    30:  ldc.i4.1
                                    31:  sub
                                    32:  stloc.0
   }                                33:  br -22 (=16)
}                                   38:  return
```

addresses

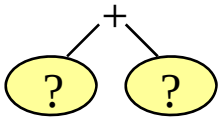
a ... 0
b ... 1
max ... 2
sum ... 3

6. Code Generation

Operands During Code Generation

Example

we want to add two values



desired code pattern

load operand 1
load operand 2
add

Depending on the operand kind we must generate different load instructions

operand kind	*instruction to be generated*
• constant	ldc.i4 x
• method argument	ldarg.s a
• local variable	ldloc.s a
• global variable	ldsfld T_{fld}
• object field	getfield a
• array element	ldelem
• loaded value on the stack	---

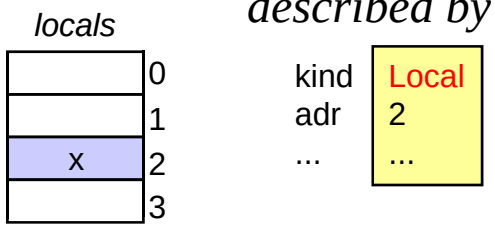
We need a descriptor, which gives us all the necessary information about operands.

Items

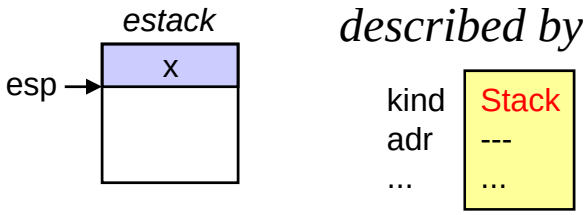
Descriptors holding the kind and the location of operands.

Example

Local variable *x* in locals area of a method state



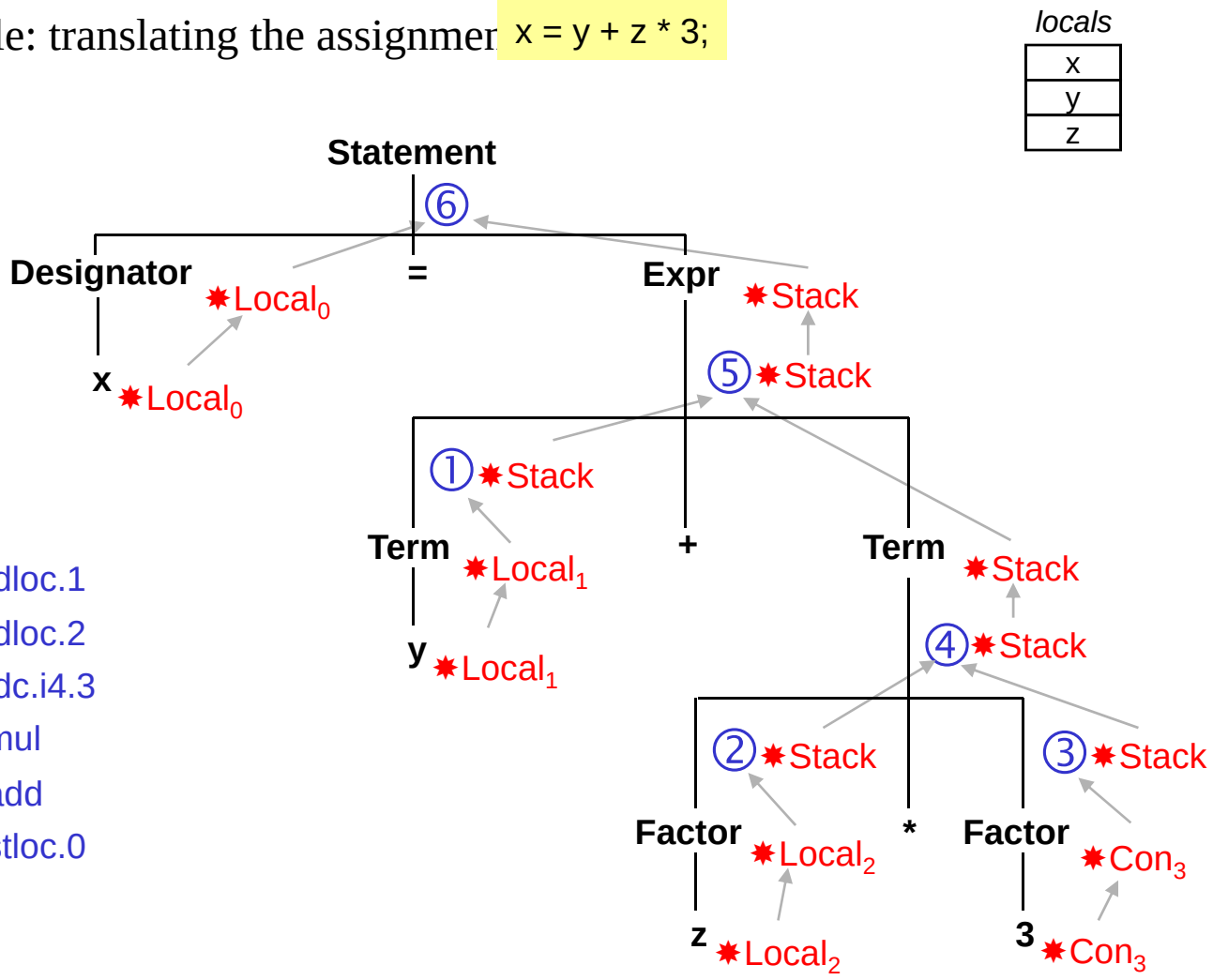
After loading the value with *ldloc.2* it is on *estack*



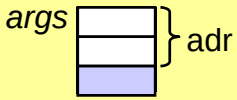
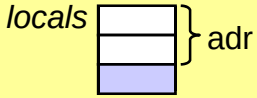
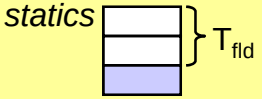
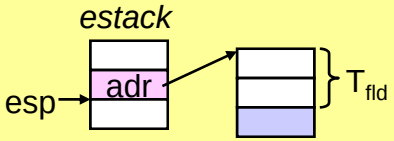
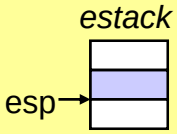
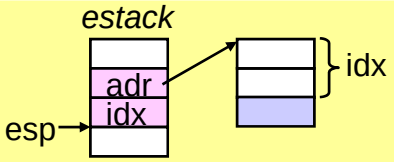
Example: Processing of Items

Most parsing methods return items (as a result of their translation work)

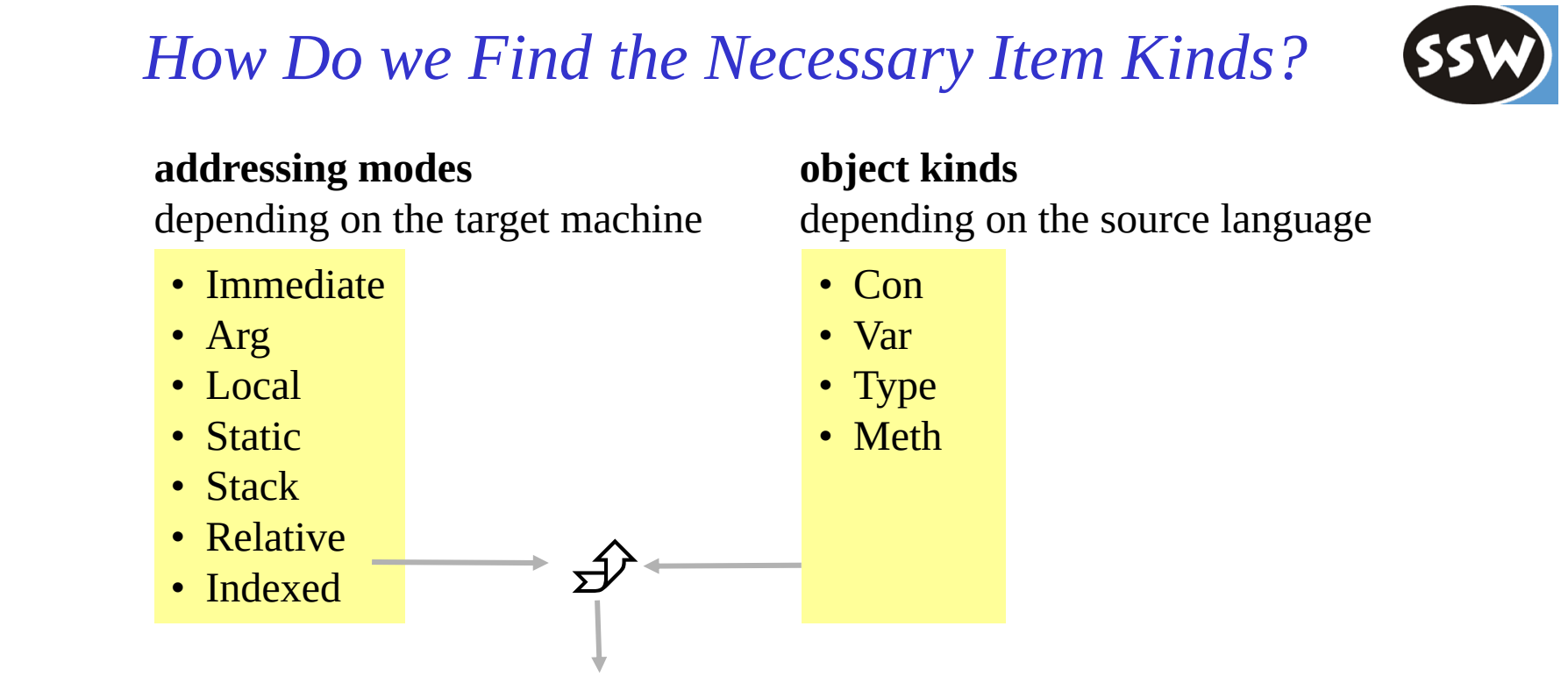
Example: translating the assignmer x = y + z * 3;



Item Kinds

operand kinds	*item kind*	*info about operands*	
constant	**Const**	constant value	
argument	**Arg**	address	
local variable	**Local**	address	
global variable	**Static**	field symol node	
object field	**Field**	field symbol node	
value on stack	**Stack**	---	
array element	**Elem**	---	
method	**Meth**	method symbol node	

How Do we Find the Necessary Item Kinds?



We do not need *Type* items in Z#, because types do not occur as operands (no type casts)

Class Item


```
class Item {
   enum Kinds { Const, Arg, Local, Static, Stack, Field, Elem, Meth }

   Kinds    kind;
   Struct   type;   // type of the operands
   int      val;    // Const: constant value
   int      adr;    // Arg, Local: address
   Symbol   sym;    // Field, Meth: Symbol node from symbol table
}
```

Constructors for creating Items

```
public Item (Symbol sym) {
   type = sym.type; val = sym.val; adr = sym.adr; this.sym = sym;
   switch (sym.kind) {
      case Symbol.Kinds.Const: kind = Kinds.Const; break;
      case Symbol.Kinds.Arg: kind = Kinds.Arg; break;
      case Symbol.Kinds.Local: kind = Kinds.Local; break;
      case Symbol.Kinds.Global: kind = Kinds.Static; break;
      case Symbol.Kinds.Field: kind = Kinds.Field; break;
      case Symbol.Kinds.Meth: kind = Kinds.Meth; break;
      default: Parser.Error("cannot create Item");
   }
}
```

create an Item from a symbol table node

```
public Item (int val) {
   kind = Kinds.Const; type = Tab.intType; this.val = val;
}
```

creates an Item from a constant

Loading Values (1)

given: a value described by an item (Const, Arg, Local, Static, ...)
wanted: load the value onto the expression stack

```
public static void Load (Item x) { // method of class Code
   switch (x.kind) {
      case Item.Kinds.Const:
         if (x.type == Tab.nullType) il.Emit(LDNULL);
         else LoadConst(x.val);
         break;
      case Item.Kinds.Arg:
         switch (x.adr) {
               case 0: il.Emit(LDARG0); break;
               ...
               default: il.Emit(LDARG, x.adr); break;
         }
         break;
      case Item.Kinds.Local:
         switch (x.adr) {
               case 0: il.Emit(LDLOC0); break;
               ...
               default: il.Emit(LDLOC, x.adr); break;
         }
         break;
      ...
```

Case analysis

depending on the item kind we have to generate different load instructions

Loading Values (2)

```
public static void Load (Item x) {  // cont.
        ...
        case Item.Static:
            if (x.sym.fld != null) il.Emit(LDSFLD, x.sym.fld); break;
        case Item.Stack: break; // nothing to do (already loaded)
        case Item.Field:  // assert: object base address is on stack
            if (x.sym.fld != null) il.Emit(LDFLD, x.sym.fld); break;
        case Item.Elem: // assert: array base address and index are on stack
            if (x.type == Tab.charType) il.Emit(LDELEMCHR);
            else if (x.type == Tab.intType) il.Emit(LDELEMINT);
            else if (x.type.kind == Struct.Kinds.Class) il.Emit(LDELEMREF);
            else Parser.Error("invalid array element type");
            break;
        default: Error("cannot load this value");
    }
    x.kind = Item.Kinds.Stack;
}
```

resulting item is always a *Stack* item

```
internal static void LoadConst (int n) {  // method of class Code
    switch (n) {
        case -1: il.Emit(LDCM1); break;
        case 0: il.Emit(LDC0); break;
        ...
        default: il.Emit(LDC, n); break;
    }
}
```

Example: Loading a Constant

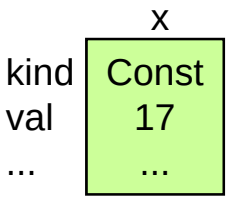
Description by an ATG

```
Factor <✸Item x>
=  number                (.  x = new Item(token.val);  // x.kind = Const
                             Code.Load(x);              // x.kind = Stack
                          .)
```

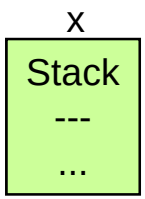
Visualisation

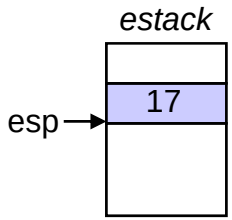
val 17

x = new Item(token.val);



Code.Load(x);





output

ldc.i4 17

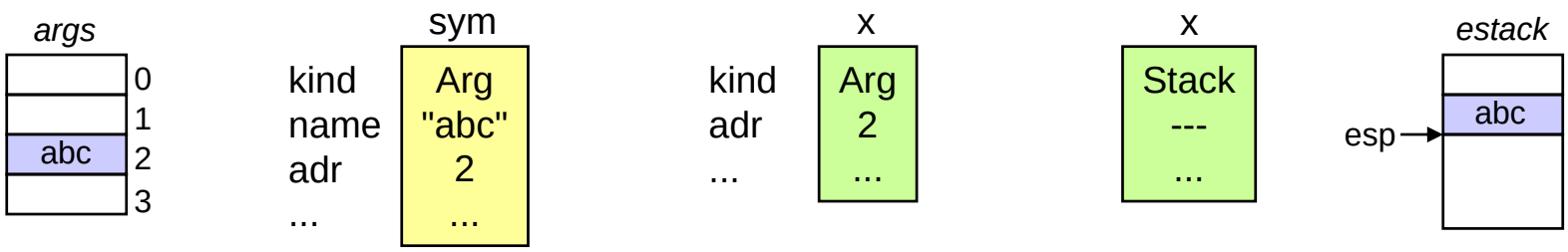
Example: Loading an Argument

Description by an ATG

```
Designator<✸Item x>
=  ident                (.  Symbol sym = Tab.Find(token.str);  // sym.kind = Const | Arg | Local | Global
                            x = new Item(sym);                 // x.kind = Const | Arg | Local | Static
                            Code.Load(x);                      // x.kind = Stack
                         .) .
```

Visualisation

sym = Tab.Find(token.str); x = new Item(sym); Code.Load(x);



output

ldarg.2

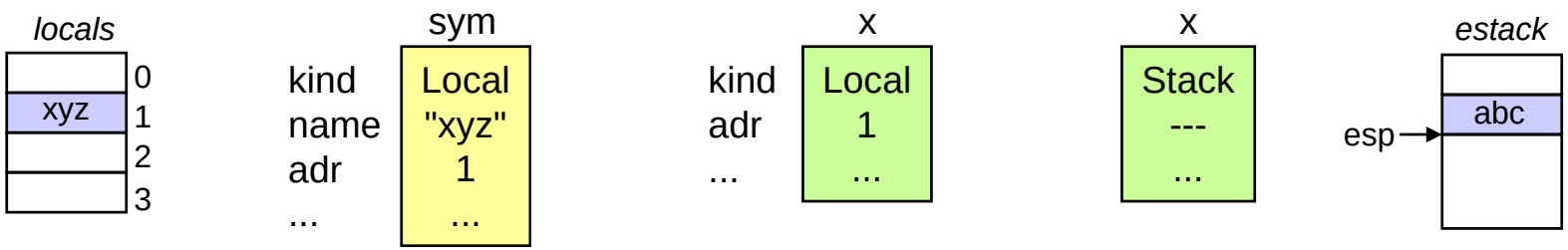
Example: Loading a Local Variable

Description by an ATG

```
Designator<✸Item x>
=  ident            (.  Symbol sym = Tab.Find(token.str);  // sym.kind = Const | Arg | Local | Global
                        Item x = new Item(sym);             // x.kind = Const | Arg | Local | Static
                        Code.Load(x);                       // x.kind = Stack
                     .) .
```

Visualisation

sym = Tab.Find(token.str); x = new Item(sym); Code.Load(x);



output

ldloc.1

Loading Object Fields

var.f

Context conditions (make sure that your compiler checks them)

Designator = Designator "." ident .

- The type of *Designator* must be a class.
- *ident* must be a field of *Designator*.

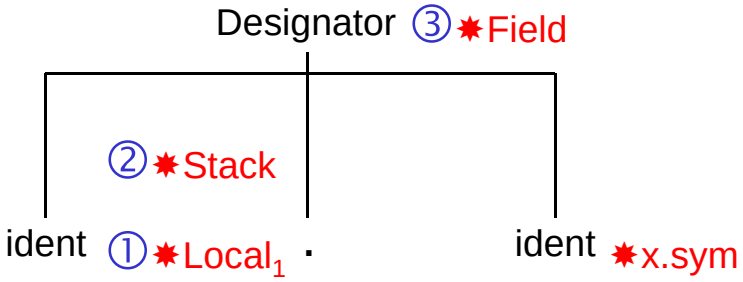
Description by an ATG

```
Designator<✸Item x>(. string name; .)
=  ident                    (. name = token.str;
                              Item x = new Item(Tab.Find(name)); .)
   { "." ident              (. if (x.type.kind == Struct.Kinds.Class) {
                                 Code.Load(x);
                                 x.sym = Tab.FindField(token.str, x.type);
                                 x.type = x.sym.type;
                               } else Error(name + " is not an object");
                               x.kind = Item.Kinds.Field;
                            .)
   | ...
   }.
```

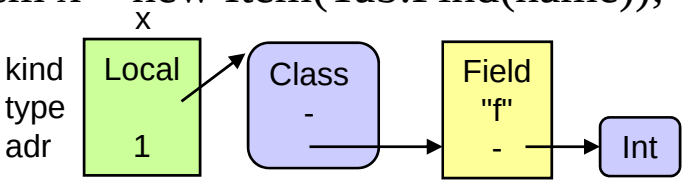
looks up *token.str* in the field list of *x.type*

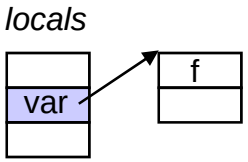
creates a *Field* item

Example: Loading an Object Field

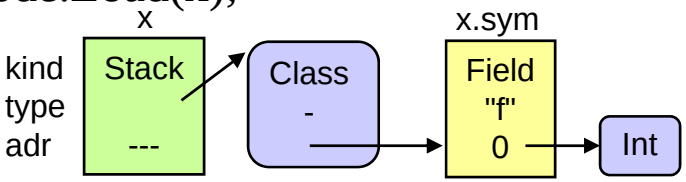


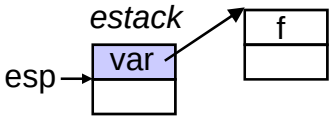
① Item x = new Item(Tab.Find(name));





② Code.Load(x);

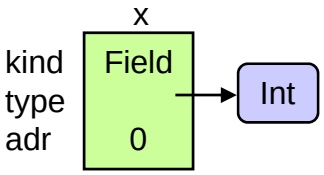


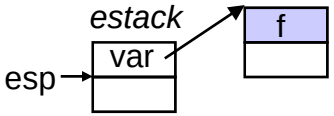


output

ldloc.1

③ create from *x* and *x.sym* a *Field* item





Loading Array Elements

a[i]

Context conditions

Designator = Designator "[" Expr "]" .

- The type of *Designator* must be an array.
- The type of *Expr* must be *int*.

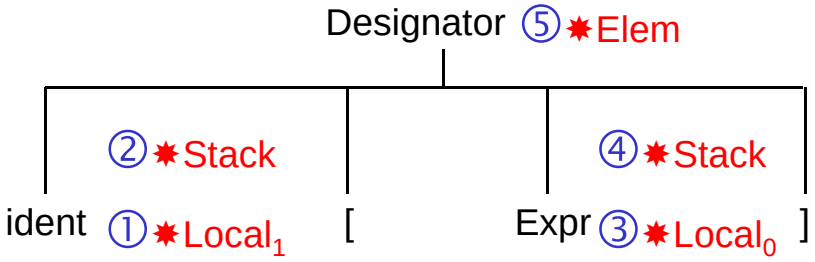
Description by an ATG

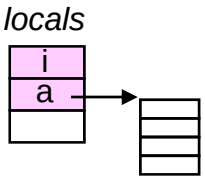
```
Designator<✸Item x>     (.  string name; Item x, y; .)
=  ident                (.  name = token.str; x = new Item(Tab.find(name)); .)
   [
      "["               (.  Code.Load(x); .)
      Expr<✸y>          (.  if (x.type.kind == Struct.Arr) {
                              if (y.type != Tab.intType) Error("index must be of type int");
                              Code.Load(y);
                              x.type = x.type.elemType;
                          } else Error(name + " is not an array");
                          x.kind = Item.Elem;
                       .)
      "]"
   ] { … }.
```

create an *Elem* item

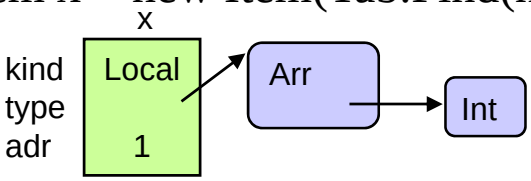
index check is done in the CLR

Example: Loading an Array Element

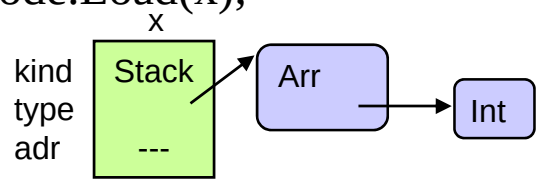




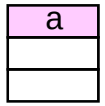
① Item x = new Item(Tab.Find(name));



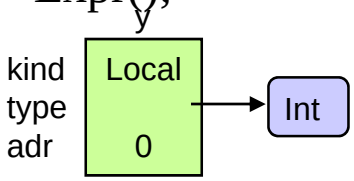
② Code.Load(x);



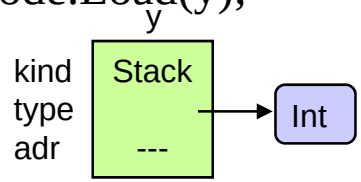
ldloc.1



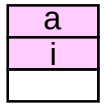
③ y = Expr();



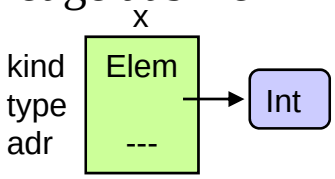
④ Code.Load(y);



ldloc.0



⑤ erzeuge aus *x* ein *Elem*-Item



6. Code Generation

Compiling Expressions

Scheme for x + y + z

```
load x
load y
add
load z
add
```

Context conditions

Expr = "-" Term.

- *Term* must be of type *int*.

Expr = Expr Addop Term.

- *Expr* and *Term* must be of type *int*.

Description by an ATG

```
Expr<✸Item x>          (.  Item y; OpCode op; .)
=  (   "-" Term<✸x>    (.  if (x.type != Tab.intType) Error("operand must be of type int");
                           if (x.kind == Item.Kinds.Const) x.val = -x.val;
                           else { Code.Load(x); Code.il.Emit(Code.NEG); }
                        .)
   |   Term<✸x>
   )
   {   ( "+"           (. op = Code.ADD; .)
       | "-"           (. op = Code.SUB; .)
       )               (.  Code.Load(x); .)
       Term<✸y>        (.  Code.Load(y);
                           if (x.type != Tab.intType || y.type != Tab.intType)
                               Error("operands must be of type int");
                           Code.il.Emit(op);
                        .)
   }.
```

Compiling Terms

Term = Term Mulop Factor.

- *Term* and *Factor* must be of type *int*.

```
Term<✸Item x>          (.  Item y; OpCode op; .)
=  Factor<✸x>
    {  ( "*"           (.  op = Code.mul; .)
       | "/"           (.  op = Code.div; .)
       | "%"           (.  op = Code.rem; .)
       )               (.  Code.Load(x); .)
       Factor<✸y>      (.  Code.Load(y);
                          if (x.type != Tab.intType || y.type != Tab.intType)
                             Error("operands must be of type int");
                          Code.il.Emit(op);
                       .)
    }.
```

Compiling Factors

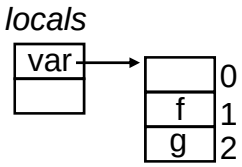
Factor = "new" ident.
• *ident* must denote a class.

Factor = "new" ident "[" Expr "]".
• *ident* must denote a type. • The type of *Expr* must be *int*.

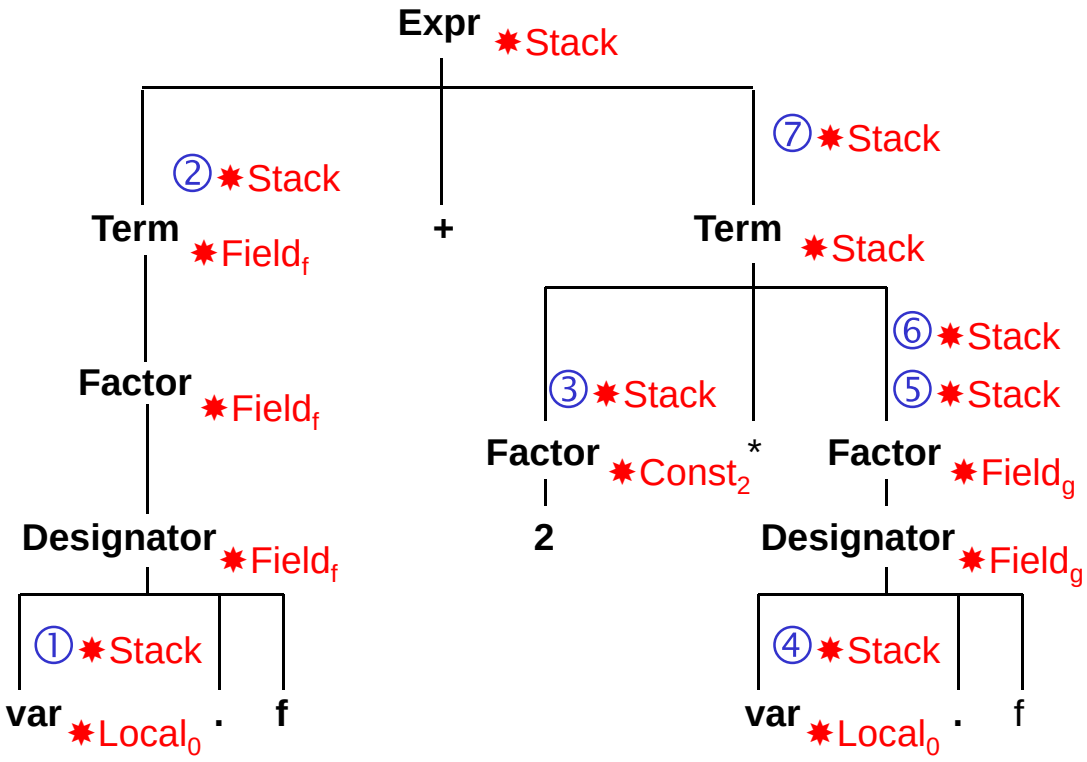
```
Factor<✸Item x>
=  Designator<✸x>          // functions calls see later
|  number                  (.  x = new Item(token.val); .)
|  charConst               (.  x = new Item(token.val); x.type = Tab.charType; .)
|  "(" Expr<✸x> ")"

|  "new" ident             (.  Symbol sym = Tab.find(token.str);
                               if (sym.kind != Symbol.Kinds.Type) Error("type expected");
                               Struct type = sym.type;
                            .)
   ( "[" Expr<✸x> "]"      (.  if (x.type != Tab.intType) Error("array size must be of type int");
                               Code.Load(x);
                               Code.il.Emit(Code.NEWARR, type.sysType);
                               type = new Struct(Struct.Kinds.Arr, type);
                            .)
   |                       (.  if (type.kind == Struct.Kinds.Class)
                                  Code.il.Emit(Code.NEWOBJ, sym.ctor);
                               else { Error("class type expected"); type = Tab.noType; }
                            .)
   )                       (.  x = new Item(Item.Kinds.Stack); x.type = type; .)
.
```

Example

var.f + 2 * var.g



① ldloc.0
② ldfld T_{fld_f}
③ ldc.i4.2
④ ldloc.0
⑤ ldfld T_{fld_g}
⑥ mul
⑦ add



6. Code Generation

Code Patterns for Assignments

5 cases depending on the kind of the designator on the left-hand side

blue instructions are generated by *Designator*!

arg = expr;	localVar = expr;	globalVar = expr;	obj.f = expr;	a[i] = expr;
... *load expr* ... **starg arg**	... *load expr* ... **stloc localVar**	... *load expr* ... **stsfld $T_{globalVar}$**	**ldloc obj** ... *load expr* ... **stfld T_f**	**ldloc a** **ldloc i** ... *load expr* ... **stelem.i2** **i4** **ref**

depending on the element type (char, int, object reference)

Compiling Assignments

Context condition

Statement = Designator "=" Expr ";".

- *Designator* must denote a variable, an array element or an object field.
- The type of *Expr* must be assignment compatible with the type of *Designator*.

Description by an ATG

```
Assignment              (.  Item x, y; .)
=  Designator<✸x>       // this call may already generate code
   "=" Expr<✸y>         (.  Code.Load(y);
                            if (y.type.AssignableTo(x.type))
                                Code.Assign(x, y);  // x: Arg | Local | Static | Field | Elem
                            else Error("incompatible types in assignment");
                         .)
   ";".
```

Assignment compatibility

y is assignment compatible with *x*, if

- *x* and *y* have the same types (*x.type* == *y.type*), or
- *x* and *y* are arrays with the same element types, or
- *x* has a reference type (class or array) and *y* is *null*

6. Code Generation

Conditional and Unconditional Jumps

Unconditional jumps

```
br offset
```

Conditional jumps

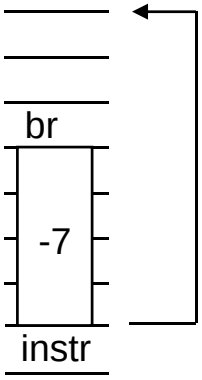
```
... load operand1 ...
... load operand2 ...
beq offset
```

if (operand1 == operand2) br offset

beq	jump on equal
bge	jump on greater or equal
bgt	jump on greater than
ble	jump on less or equal
blt	jump on less than
bne.un	jump on not equal

Forward and Backward Jumps

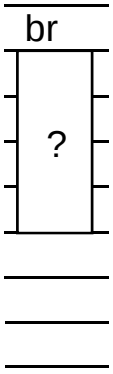
Backward jumps



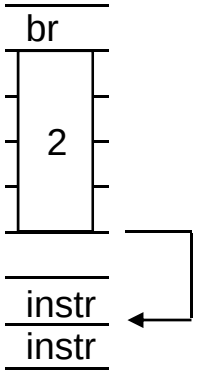
target address (label) is already known
(because the instruction at this position has already been generated)

jump distance
- 4 bytes long (CIL also has short form with 1 byte addresses
- relative to the beginning of the next instruction (= end of jump instructio

Forward jumps



target address still unknown
- leave it empty
- remember "fixup address"



patch it when the target address becomes known (fixup)

Struct System.Reflection.Emit.Label

Representation of Jump Labels

```
struct Label { ... }
```

Labels are managed by *ILGenerator*.

```
class ILGenerator {
   ...
   Label DefineLabel ();          // create a yet undefined jump label
   void MarkLabel (Label);        // define the label at the current position in the IL stream
}
```

Use

```
Label label = Code.il.DefineLabel();
...
Code.il.Emit(Code.BR, label);    // jump to the yet undefined label (forward jump)
..
Code.il.MarkLabel(label);        // now the branch to label leads here
```

Conditions

Conditions

```
if ( a > b ) ...
```

Condition

code pattern

```
load a
load b
ble ...
```

- *Condition* returns the compare operator;
  the comparison is then done in the jump instruction

Cond item

Condition returns an item of a new kind *Item.Kinds.Cond* with the following contents:

- token code of compare operator:

```
public int relop;  // Token.EG, .GE, .GT, .LE, .LT, .NE
```

- jump labels for jumps that lead out of Condition
  - public Label **tLabel** : for true-jumps
  - public Label **fLabel** : for false-jumps

} necessary for conditions, which contain&& and || operators

True-jumps and false-jumps

true-jump jump if the condition is true — a > b ⇩ bgt ...

false-jump jump if the condition is false — a > b ⇩ ble ...

Cond Items


```
class Item {
   public enum Kinds { Const, Arg, Local, Static, Field, Stack, Elem, Meth, Cond }

   Kinds    kind;
   Struct   type;      // type of the operand
   int      val;       // Const: constant value
   int      adr;       // Arg, Local: address
   int      relop;     // Cond: token code of the operator
   Label    tLabel;    // jump label for true jumps
   Label    fLabel;    // jump label for false jumps
   Symbol   sym;       // Field, Meth: symbol table node
}
```

New constructor (for *Cond* items)

```
public Item (int relop, Struct type) {
   this.kind = Kinds.Cond;
   this.type = type;
   this.relop = relop;
   tLabel = Code.il.DefineLabel();
   fLabel = Code.il.DefineLabel();
}
```

Generating Conditional Jumps

With methods of class *Code*

```
class Code {
   static readonly OpCode[] brTrue = { BEQ, BGE, BGT, BLE, BLT, BNE };
   static readonly OpCode[] brFalse = { BNE, BLT, BLE, BGT, BGE, BEQ };
   ...
   internal static void TJump (Item x) {
      il.Emit(brTrue[x.relop - Token.EQ], x.tLabel);
   }

   internal static void FJump (Item x) {
      il.Emit(brFalse[x.relop - Token.EQ], x.fLabel);
   }
   ...
}
```

Use

```
"if" "(" Condition<✸x> ")"          (.  Code.FJump(x); .)
```

Generating Unconditional Jumps

With a method of class *Code*

```
class Code {
   ...
   internal static void Jump (Label lab) {
      il.Emit(BR, lab);
   }
   ...
}
```

Use

```
Label label = Code.il.DefineLabel();
...
Code.Jump(label);
```

6. Code Generation

while Statement

Desired code pattern

```
while
(Condition)

Block
```

```
top:
        ... code for Condition ...
        FJump end
        ... code for Block ...
        br top
end:    ...
```

Description by an ATG

```
WhileStatement                  (.  Item x; .)
=  "while"                      (.  Label top = Code.il.DefineLabel();
                                    top.here();
                                .)
   "(" Condition<✸x> ")"        (.  Code.FJump(x); .)
   Block                        (.  Code.Jump(top);
                                    Code.il.MarkLabel(x.fLabel);
                                .)
   .
```

Example

```
while (a > b) a = a - 2;

10  ldloc.0      <-- top
11  ldloc.1
12  ble 9 (=26)
17  ldloc.0
18  ldc.i4.2
19  sub
20  stloc.0
21  br -16 (=10)
26  ...          <-- x.fLabel
```

if Statement

Desired code pattern

```
if
(Condition)

Block
```

```
        ... Condition ...
        FJump end
        ... Block ...
end:    ...
```

```
if
(Condition)

Block

else
Block
```

```
        ... Condition ...
        FJump else
        ... Block ...
        br end
else:
        ... Block ...
end:    ...
```

Description by an ATG

```
IfStatement                    (.  Item x; Label end; .)
=  "if"
   "(" Condition<✸x> ")"       (.  Code.FJump(x); .)
   Block
   (   "else"                  (.  end = Code.il.DefineLabel();
                                   Code.Jump(end);
                                   Code.il.MarkLabel(x.fLabel);
                                .)
       Block                   (.  Code.il.MarkLabel(end); .)
   |                           (.  Code.il.MarkLabel(x.fLabel); .)
   ).
```

Example

```
if (a > b) max = a; else max = b;
```

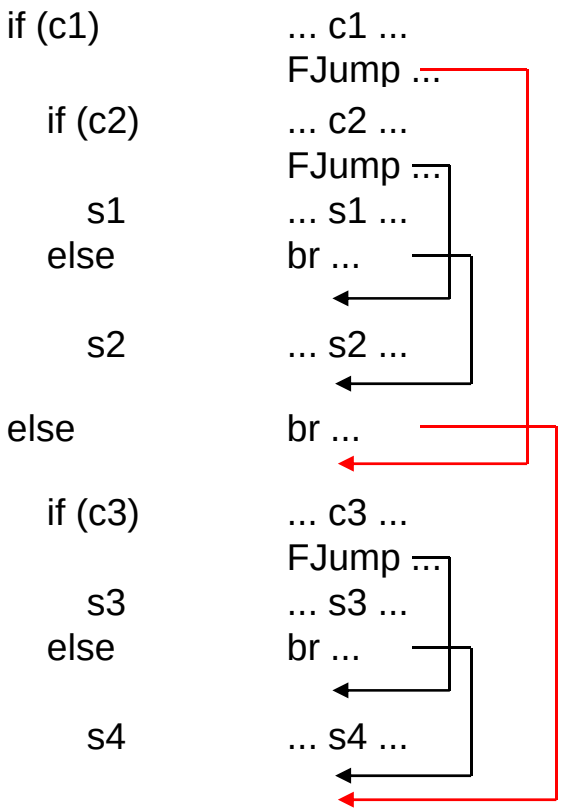
```
10   ldloc.0
11   ldloc.1
12   ble 7 (=24)
17   ldloc.0
18   stloc.2
19   br 2 (=26)
24   ldloc.1
25   stloc.2
26   ...
```

x.fLabel

end[249]

Works Also for Nested ifs

```
IfStatement                    (.  Item x; Label end; .)
=  "if"
   "(" Condition<✸x> ")"       (.  Code.FJump(x); .)
   Block
   (   "else"                  (.  end = Code.il.DefineLabel();
                                   Code.Jump(end);
                                   Code.il.MarkLabel(x.fLabel);
                               .)
       Block                   (.  Code.il.MarkLabel(end); .)
   |                           (.  Code.il.MarkLabel(x.fLabel); .)
   ).
```



break Statement

For jumping out of a loop

- define a label *breakLab* at the end of the loop
- if a break occurs in the loop: Code.Jump(breakLab);

Nested loops

- every loop needs its own *newBreakLab*
- *newBreakLab* must be passed to nested statements as an attribute

```
Statement<✵Label breakLab>
=  "while"                          (.  Label newBreakLab = Code.il.DefineLabel(); ... .)
   "(" Condition<✸x> ")"            (. ... .)
   Block<✵newBreakLab>              (. Code.il.MarkLabel(newBreakLab); .)

|  "{" { Block<✵breakLab> } "}"

|  "if"
   ...
    Block<✵breakLab>
   ...

|  "break"                          (.  if (breakLab.Equals(undef)) Error("break outside a loop");
                                       Code.Jump(breakLab);
                                    .)

|  ... .
```

```
static readonly Label undef;
```

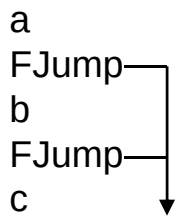
because structs cannot be null (yet)

Question: What would we have to do in order to use break with a label name?

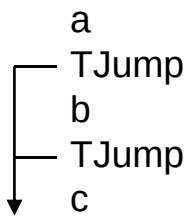
Short-Circuit Evaluation of Boolean Expressions

- Boolean expression may contain && and || operators
- The evaluation of a Boolean expression stops as soon as its result is known

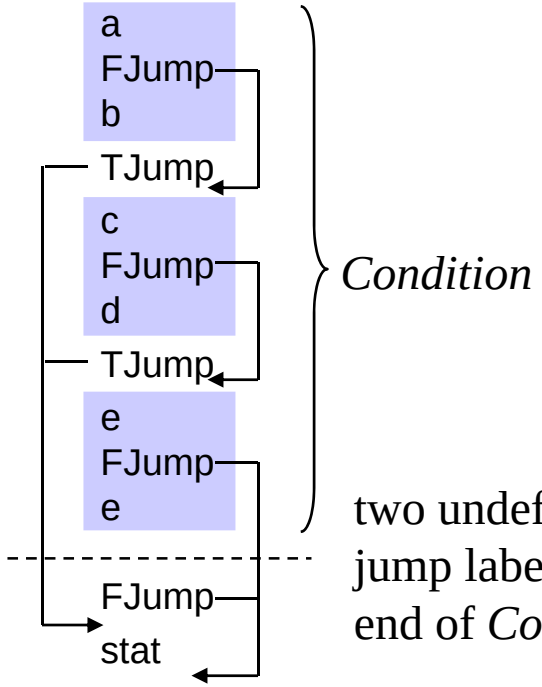
a && b && c



a || b || c



if (a && b || c && d || e && f) stat;



Needs a small change in the ATG of if statements

```
IfStatement
= "if"
   "(" Condition<✸x> ")"        (.  Code.FJump(x);
                                    Code.il.MarkLabel(x.tLabel);
                                 .)
   Block                        (.  Code.il.MarkLabel(x.fLabel); .)
   .
```

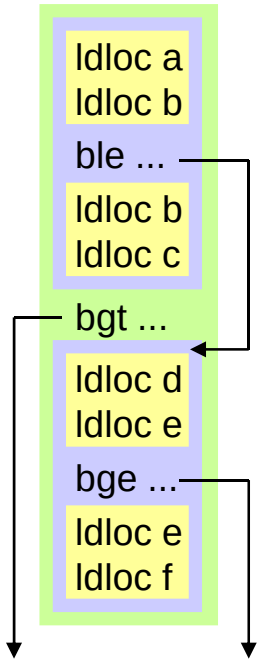
Compiling Boolean Expressions

```
CondFactor<✸Item x> (. Item y; int op; .)
=  Expr<✸x>         (. Code.Load(x); .)
   Relop<✸op>
   Expr<✸y>         (. Code.Load(y);
                       if (!x.type.CompatibleWith(y.type))
                          Error("type mismatch");
                       else if (x.type.IsRefType() &&
                                op!=Token.EQ && op!=Token.NE)
                          Error("only equality checks ...");
                       x = new Item(op, x.type);
                    .)
   .
```

```
CondTerm<✸Item x>    (. Item y; .)
=  CondFactor<✸x>
   {  "&&"           (. Code.FJump(x); .)
      CondFactor<✸y> (. x.relop = y.relop; x.tLabel = y.tLabel; .)
   }.
```

```
Condition<✸Item x>   (. Item y; .)
=  CondTerm<✸x>
   {  "||"           (. Code.TJump(x);
                        Code.il.MarkLabel(x.fLabel);
                     .)
      CondTerm<✸y>   (. x.relop = y.relop; x.fLabel = y.fLabel; .)
   }.
```

a>b && b>c || d<e && e<f



6. Code Generation

Procedure Call

Code pattern

M(a, b);	ldloc a ldloc b call T_{meth}	parameters are passed on the *estack*

Description by an ATG

```
Statement                    (.  Item x, y; ... .)
=  Designator<✸x>
   (   ActParList<❋x>        (.  Code.il.EmitCall(Code.CALL, x.sym.meth, null);
                                  if (x.type != Tab.noType) Code.il.Emit(Code.POP);
                                .)
   |   "=" Expr<✸y> ";"       (. ... .)
   )
| ... .
```

Function Call

Code pattern

c = M(a, b);	ldloc.s a ldloc.s b call T_{meth} stloc.s c	parameters are passed on the *estack* function value is returned on the *estack*

Standard functions

ord('x')

- *ActParList* loads 'x' onto the *estack*
- the loaded value gets the type of *ordSym* (= *intType*) and *kind* = *Item.Kinds.Stac*

Description by an ATG

```
Factor<✸Item x>
=  Designator<✸x>
   [   ActParList<❋x>     (.  if (x.type == Tab.noType) Error("procedure called as a function");
                              if (x.sym == Tab.ordSym || x.sym == Tab.chrSym) ; // nothing to do
                              else if (x.sym == Tab.lenSym)
                                 Code.il.Emit(Code.LDLEN);
                              else if (x.kind == Item.Kinds.Meth)
                                 Code.il.Emit(Code.CALL, x.sym.meth, null);
                              x.kind = Item.Stack;
                           .)
   ]
| ... .
```

Method Declaration


```
MethodDecl                  (. Struct type; int n; .)
=  (   Type<✸type>
   |   "void"               (. type = Tab.noType; .)
   )
   ident                    (. curMethod = Tab.Insert(Symbol.Kinds.Meth, token.str, type);
                               Tab.openScope();
                            .)
   "(" [ FormPars ] ")"     (. curMethod.nArgs = Tab.topScope.nArgs;
                               curMethod.locals = Tab.topScope.locals;
                               Code.CreateMetadata(curMethod);
                               if (curMethod.name == "Main") {
                                  if (curMethod.type != Tab.noType) Error("method Main must be void");
                                  if (curMethod.nArgs != 0) Error("method Main must not have parameters");
                               }
                            .)
   { VarDecl<✵Symbol.Kinds.Local>
   }                        (. curMethod.nLocs = Tab.topScope.nLocs;
                               curMethod.locals = Tab.topScope.locals;
                            .)
   Block<✵undef>            (. if (curMethod.type == Tab.noType) Code.il.Emit(Code.RET);
                               else {  // end of function reached without a return statement
                                  Code.il.Emit(Code.NEWOBJ, Code.eeexCtor);
                                  Code.il.Emit(Code.THROW);
                               }
                               Tab.closeScope();
                            .)
   .
```

global variable

no-arg constructor of
System.ExecutionEngineException

Formal Parameters

- are entered into the symbol table (as *Arg* symbols of the method scope)
- the method scope counts their number (in *nArgs*)

```
FormPars = FormPar { "," FormPar } .
```

```
FormPar                 (.  Struct type; .)
=  Type<✸type>
   ident                (.  Tab.Insert(Symbol.Kinds.Arg, token.str, type); .)
   .
```

Actual Parameters

- load them to *estack*
- check if they are assignment compatible with the formal parameters
- check if the number of actual and formal parameters corresponds

```
ActPars<✵Item m>  (. Item ap; int aPars = 0, fPars = 0; .)
=  "("            (. if (m.kind != Item.Kinds.Meth) { Error("not a method"); m.sym = Tab.noSym; } .)
                     fPars = m.sym.nArgs;
                     Sym fp= m.sym.locals;
                   .)
   [   Expr<✸ap>   (. Code.Load(ap); aPars++;
                     if (fp != null && fp.kind == Symbol.Kinds.Arg) {
                        if (!ap.type.AssignableTo(fp.type)) Error("parameter type mismatch");
                        fp = fp.next;
                     }
                   .)
       { "," Expr<✸ap> (. Code.Load(ap); aPars++;
                     if (fp != null && fp.kind == Symbol.Kinds.Arg) {
                        if (!ap.type.AssignableTo(fp.type)) Error("parameter type mismatch");
                        fp = fp.next;
                     }
                   .)
       }
   ]               (. if (aPars > fPars) Error("more actual than formal parameters");
                     else if (aPars < fPars) Error("less actual than formal parameters");
                   .)
   ")" .
```

return Statement


```
Statement<✵Label break>
=  ...
|  "return"
   (                    (.  if (curMethod.type == Tab.noType)
                                Error("void method must not return a value");
                        .)
      Expr<✸x>          (.  Code.Load(x);
                            if (x.type.AssignableTo(curMethod.type))
                                Error("type of return value must be assignable to method type");
                        .)
   |                    (.  if (curMethod.type != Tab.noType) Error("return value expected"); .)
   )                    (.  Code.il.Emit(Code.RET); .)
    ";" .
```

7. Bottom-up Parsing

How a Bottom-up Parser Works

Example

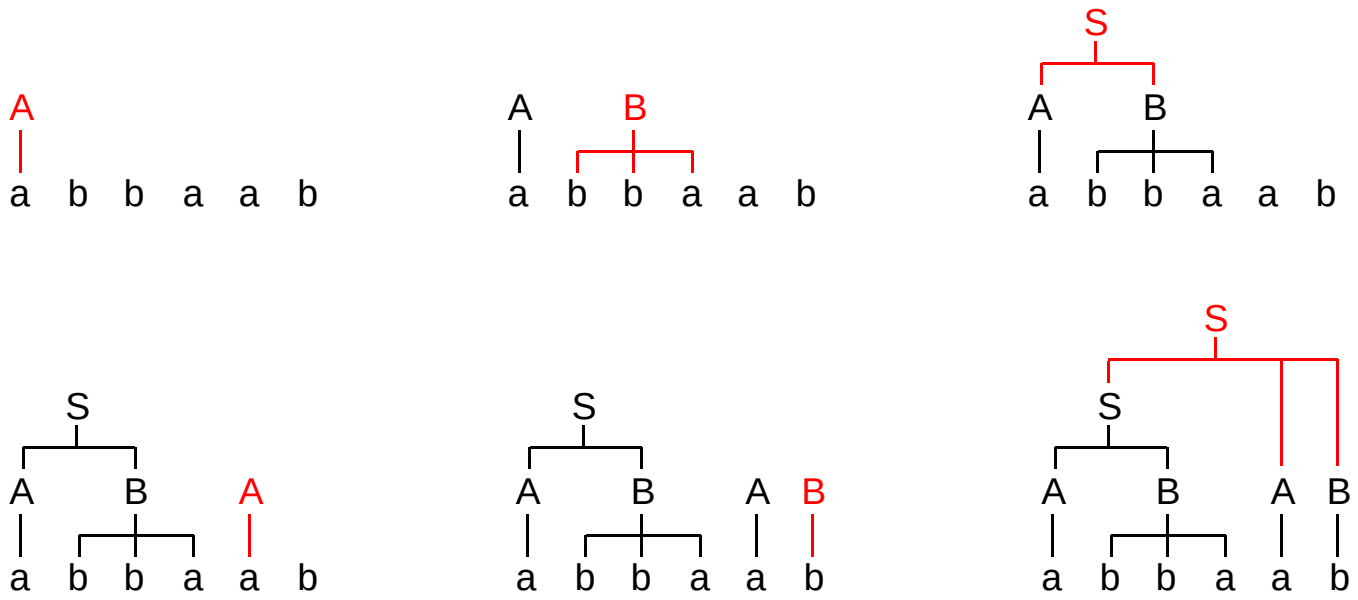
```
S = A B | S A B.
A = a | a a b.
B = b | b b a.
```

not LL(1)!

- cannot be used for top-down parsing
- but no problem for bottom-up parsing

input: a b b a a b

Syntax tree is built bottom-up



Using a Stack for Parsing

grammar

```
S = A B | S A B.
A = a | a a b.
B = b | b b a.
```

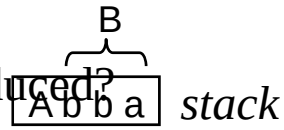
input

a b b a a b # (# ... eof token)

stack	*input*	*action*
	a b b a a b #	shift (i.e. read next token and push it on the stack)
a	b b a a b #	reduce *a* to *A*
A	b b a a b #	shift
A b	b a a b #	shift (not reduce, because that would lead to a dead end!)
A b b	a a b #	shift
A b b a	a b #	reduce *b b a* to *B*
A B	a b #	reduce *A B* to *S*
S	a b #	shift
S a	b #	reduce *a* to *A*
S A	b #	shift
S A b	#	reduce *b* to *B*
S A B	#	reduce *S A B* to *S*
S	#	sentence recognized (stack contains the start symbol; input is emp

Using a Stack for Parsing (cont.)

What does the parser need to know?



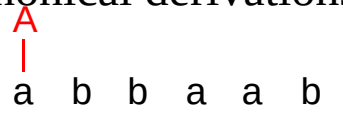
- Does the stack end contain something that can be reduced?
- To which NTS should it be reduced?
- Should the parser shift or reduce in order not to get into a dead end?

Four parsing actions

shift	read and push next input token (TS)
reduce	reduce the end of a stack to a NTS
accept	sentence recognized (only if *S* . #)
error	none of the above actions is possible (error handling)

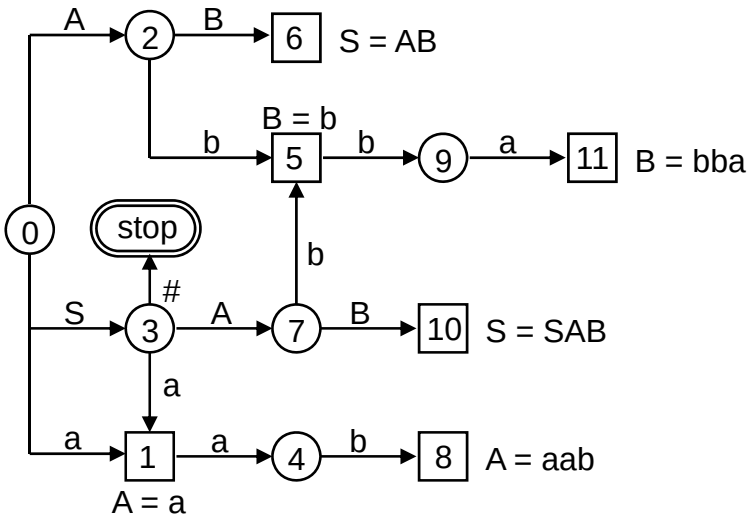
Bottom-up parsers are therefore called

- **Shift-reduce parsers**
- **LR parsers**
  they recognize sentences from **L**eft to right using **R**ight-canonical derivations
  right-canonical derivation = left-canonical reduction
  i.e. the leftmost simple phrase (the *handle*) is reduced



The Parser as a Push-down Automaton

Push-down automaton



Grammar

1	S = A B.
2	S = S A B.
3	A = a.
4	A = a a b
5	B = b
6	B = b b a

only plain BNF allowed!

State transition table (parsing table)

	TS			NTS		
	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

s1 ... shift to state 1
r3 ... reduce acording to production 3
- ... error

Parsing with Shift-Reduce Actions

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

1	S = A B.
2	S = S A B.
3	A = a.
4	A = a a b
5	B = b
6	B = b b a

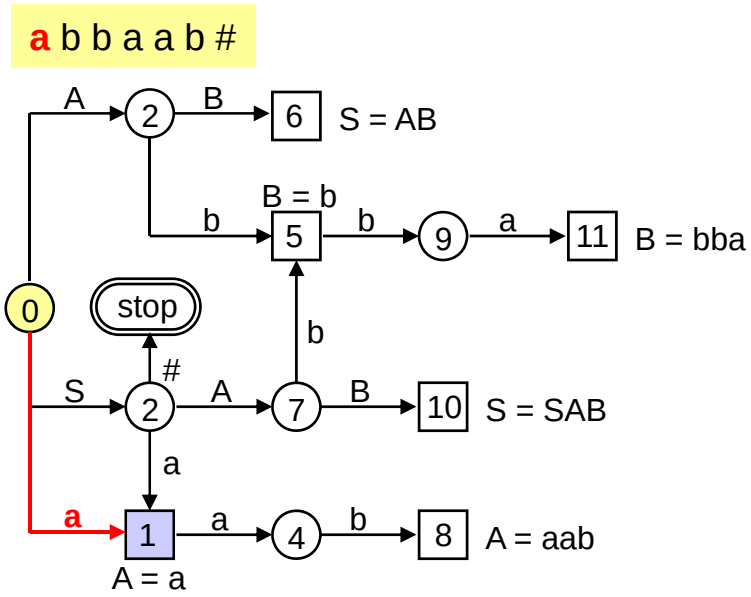
Input: a b b a a b #

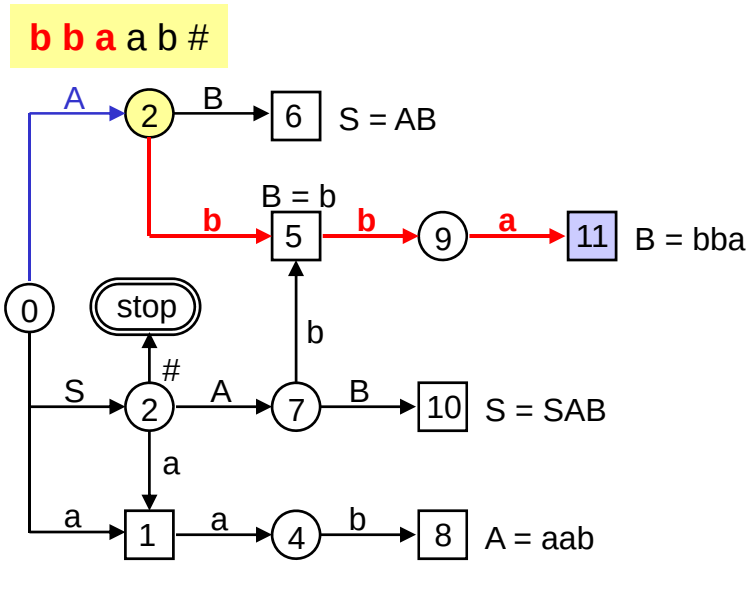
stack		*input*	*action*
0		a b b a a b #	s1
0 1		b b a a b #	r3 (A=a)
0	A	b b a a b #	s2 (shift with A!)
0 2		b b a a b #	s5
0 2 5		b a a b #	s9
0 2 5 9		a a b #	s11
0 2 5 9 11		a b #	r6 (B=bba)
0 2	B	a b #	s6
0 2 6		a b #	r1 (S=AB)
0	S	a b #	s3
0 3		a b #	s1
0 3 1		b #	r3 (A=a)
0 3	A	b #	s7
0 3 7		b #	s5
0 3 7 5		#	r5 (B=b)
0 3 7	B	#	s10
0 3 7 10		#	r2 (S=SAB)
0	S	#	s3
0 3		#	acc

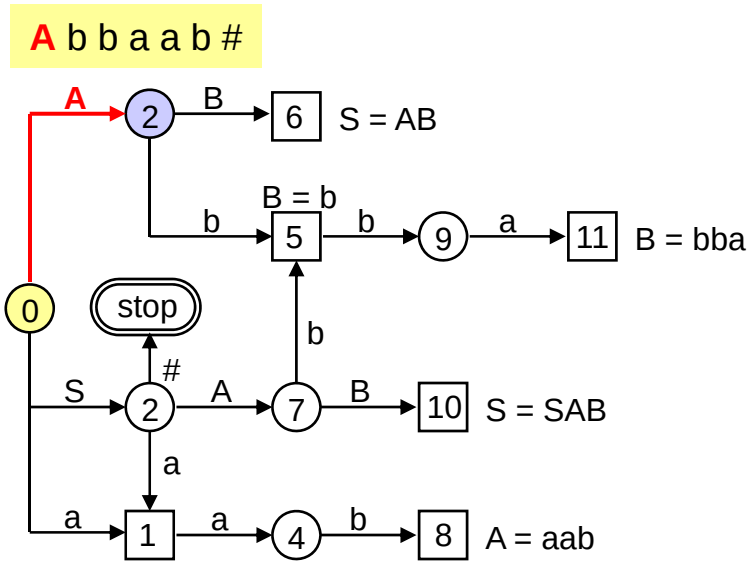
r3 ... reduce according to production 3 (A = a)

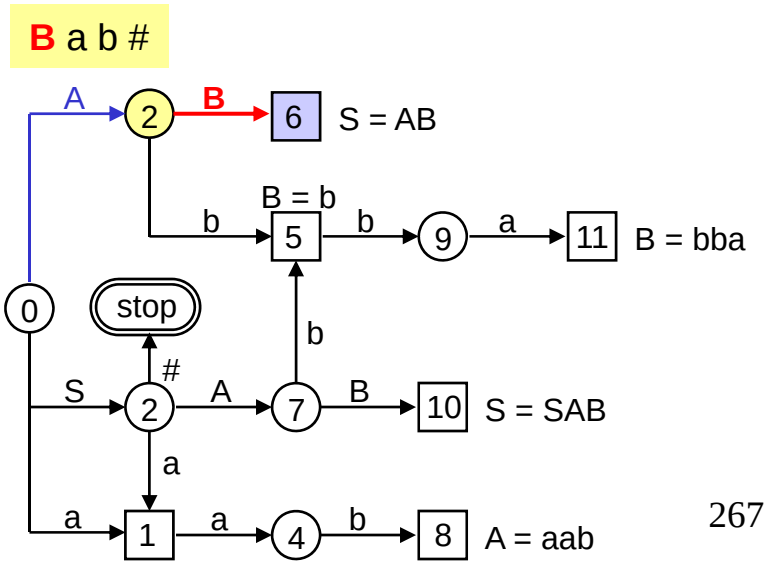
- pop 1 state (because the right-hand side has length
- insert the left-hand side (*A*) into the input stream
- reduce is always followed by a shift with a NTS

Simulating a Push-down Automaton

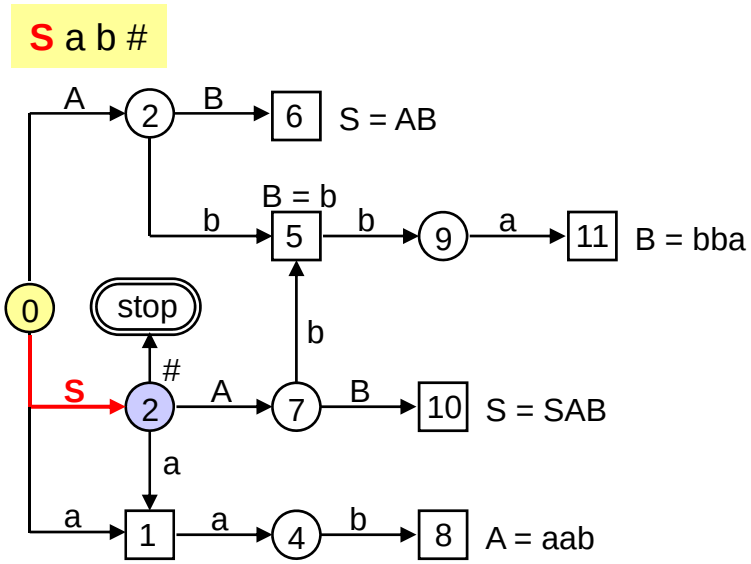


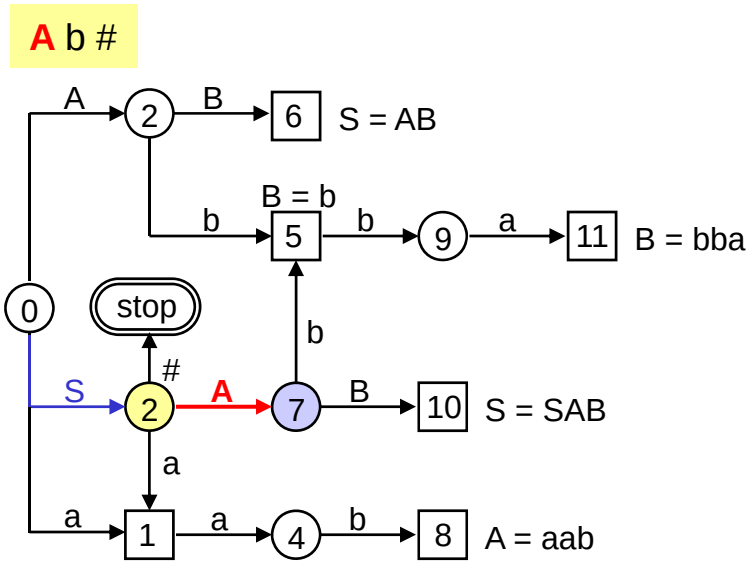


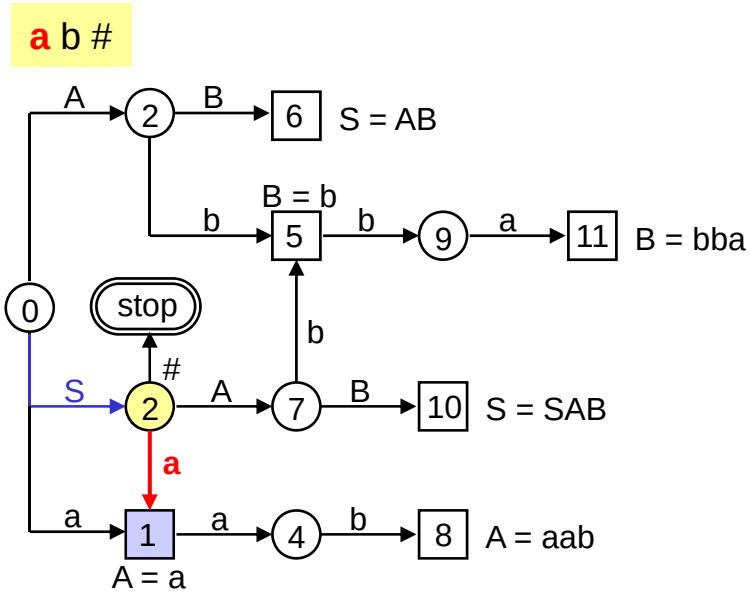


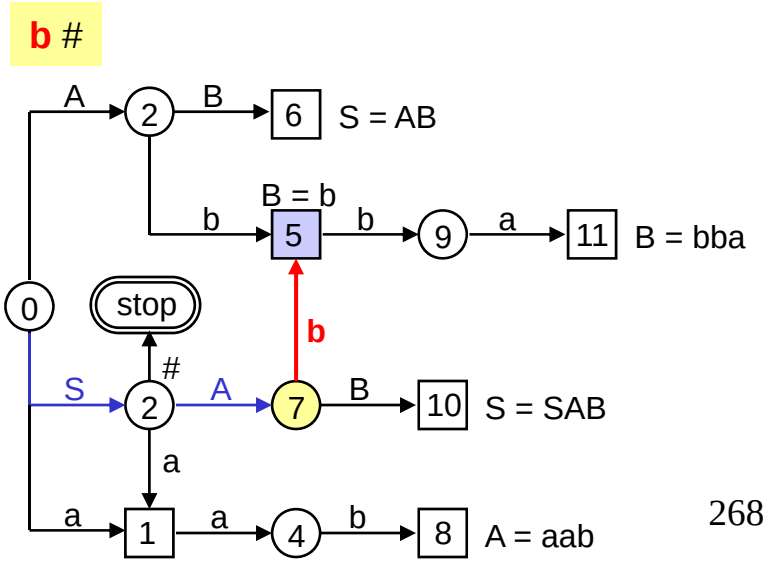


Simulating a Push-down Automaton



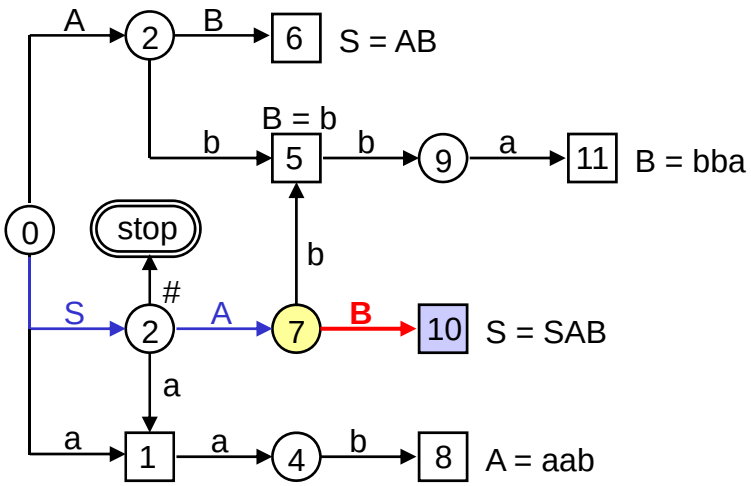


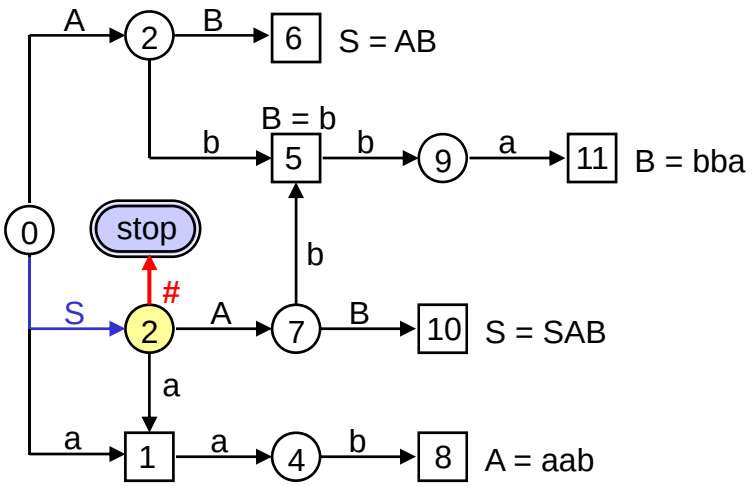




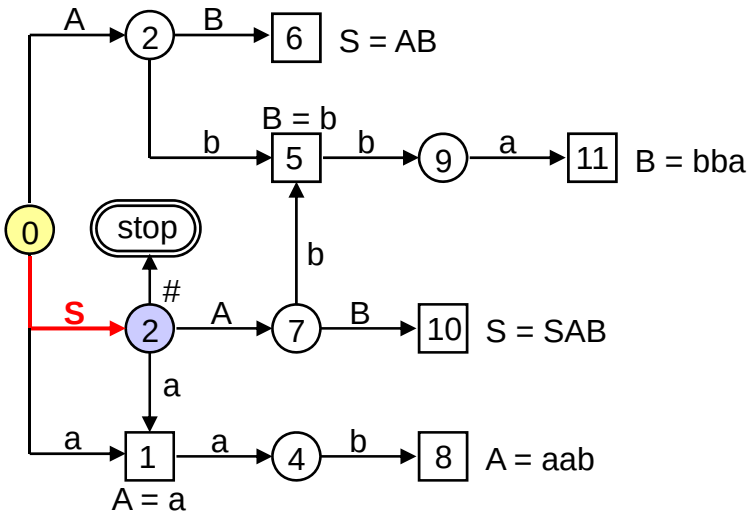
Simulating a Push-down Automaton

B #





#

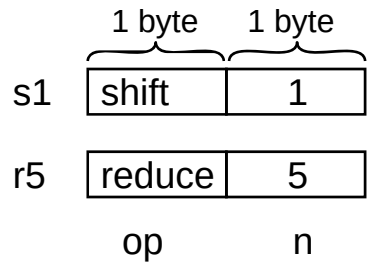


Implementation of an LR Parser


```
void Parse () {
   short[,]   action = { {...}, {...}, ...};   // state transition table
   byte[]     length = { ... };                // production lengths
   byte[]     leftSide = { ... };              // left side NTS of every production
   byte       state;                           // current state
   int        sym;                             // next input symbol
   int        op, n, a;

   ClearStack();
   state = 0; sym = Next();
   for (;;) {
      Push(state);
      a = action[state, sym]; op = a / 256; n = a % 256;
      switch (op) {
         case shift:       // shift n
            state = n; sym = Next(); break;
         case reduce:      // reduce n
            for (int i = 0; i < length[n]; i++) Pop();
            a = action[Top(), leftSide[n]]; n = a % 256;  // shift n
            state = n;
            break;
         case accept: return;
         case error: throw new Exception();  // error handling is still missing
      }
   }
}
```

table-driven program
for arbitrary grammars

action entry (2 bytes)



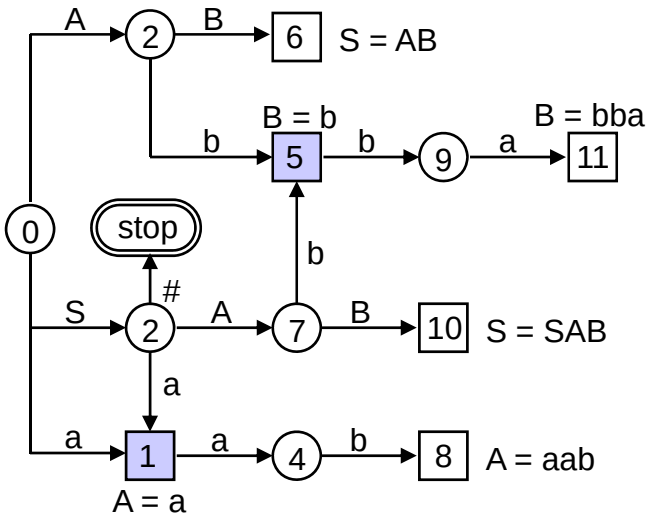
7. Bottom-up Parsing

LR(0) and LR(1) Grammars

LR(0) Parsable from **L**eft to right
with **R**ight-canonical derivations
and **0** lookahead tokens

A grammar is LR(0), if

- there is no reduce state that has also a shift action
- in every reduce state it is only possible to reduce according to a single production



This grammar is not LR(0)!

LR(1) Parsable from **L**eft to right
with **R**ight-canonical derivations
and **1** lookahead token

A grammar is LR(1), if in every state a single lookahead token is sufficient to decide

- whether to do a shift or a reduce
- according to which production one should reduce

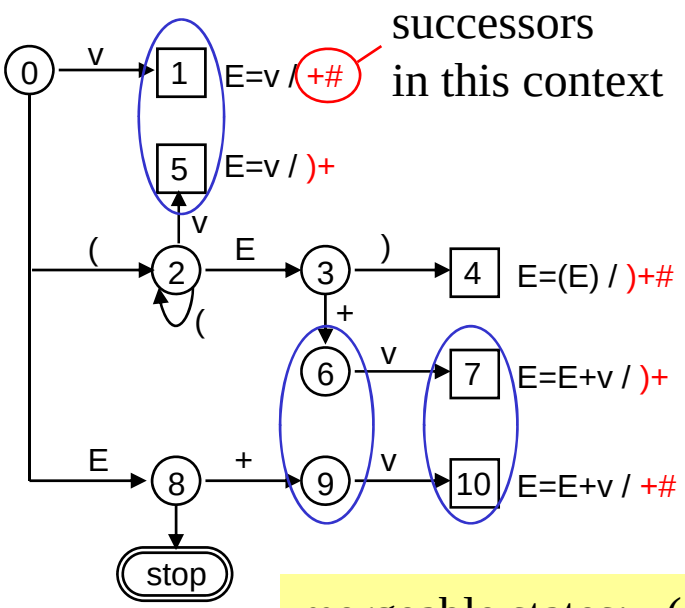
LALR(1) Grammars

Lookahead LR(1)

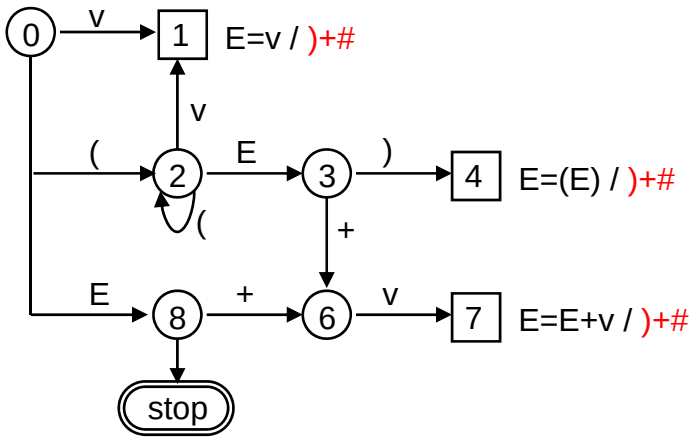
- Subset of LR(1) grammars
- Have smaller tables than LR(1) grammars, because states with the same actions but different lookahead tokens can be merged.

Example E = v | E "+" v | "(" E ")"

LR(1) tables



LALR(1) tables



mergeable states: (1,5) (6,9) (7,10)

LR Parsing Versus Recursive Descent

Advantages

- LALR(1) is more powerful than LL(1)
  - allows left-recursive productions
  - allows productions with the same terminal start symbols
- LR parsers are more compact than recursive descent parsers
  (but the tables need much memory, too)
- Table-driven parsers are universal algorithms that can be parameterized with tables
- Table-driven parsers allow better error handling

Disadvantages

- LALR(1) tables are difficult to build for large grammars (one needs tools)
- LR parsers are slightly slower than RD parsers
- Semantic processing is more complicated for LR parsers
- LR parsers are more difficult to trace

LR parsers only pay off for complex languages.
For simple languages (e.g. command languages) recursive descent parsers are better.

7. Bottom-up Parsing

LR Items

Example

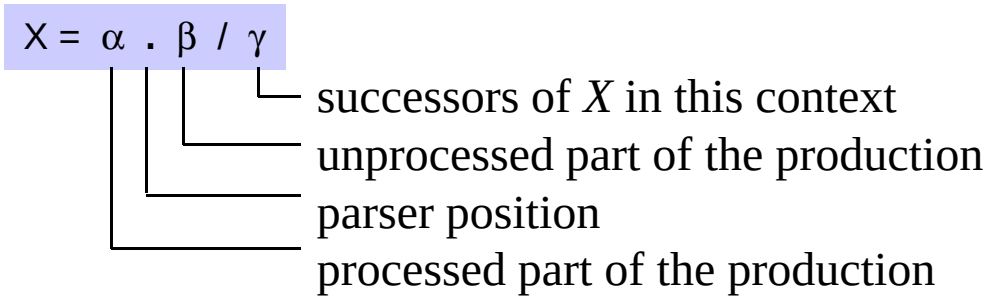
```
S = a A b
A = c
```

The parser (denoted by a point) **moves through the productions**

```
S = . a A b
S = a . A b   ←  if the parser is in front of A, it is also
                 at the beginning of the A-production
A = . c
A = c .
S = a A . b
S = a A b .
```

LR item

parsing snapshot



Control symbol

the symbol after the point, e.g.:

A = a . a b / c	control symbol = a
A = a a b . / c	control symbol = c

shift item	X = α . β / γ	point is not at the end of the production
reduce item	X = α . / γ	point is at the end of the production

Parser State as a Set of Items

Parser State

Represented by a set of items on which the parser works at this time

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

Top-down parsers always work on just *one* production.
Bottom-up parsers can work on *multiple productions in parallel*.

```
X = . a a b        shift a      X = a . a b        shift a      X = a a . b
X = . a a c   ------------->    X = a . a c   ------------->    X = a a . c
```

only at this point the parser has to decide in favor of one of the two productions (based on the lookahead token)

Bottom-up parsers are therefore more powerful than top-down parsers.

Kernel and Closure of a State

Kernel

Those items of a state which do not start with a point (except in the start symbol produc

```
S = A . B c / #
```

All other items of a state can be derived from its kernel.

Closure

```
do {
   if (state has an item of the form X = α . Y β / γ)
      add all items Y = . ω / First(β) to the state;
} while (new items were added);
```

Example

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

kernel } closure

```
S = A B c
A = a
A = a a b
B = b
B = b b a
```

Example: Computing the Closure

Grammar

```
0   S' = S #
1   S = A B
2   S = S A B
3   A = a
4   A = a a b
5   B = b
6   B = b b a
```

Kernel

```
S' = . S #
```

Add all *S* productions

```
S' = . S #
S = . A B      / #
S = . S A B    / #
```

Add all *A* productions

```
S' = . S #
S = . A B      / #
S = . S A B    / #
A = . a        / b
A = . a a b    / b
```

Add all *S* productions (because of `S = . S A B / #`)

Successors of *S* = First(*A*) = {*a*}

```
S' = . S #
S = . A B      / #a
S = . S A B    / #a
A = . a        / b
A = . a a b    / b
```

} closure

Successor State

Succ(s, sym)

The state in which we get from state *s* by doing *shift sym* (*sym* = control symbol)

Example

state *i*:

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

Succ(*i*, *b*):

```
B = b .        / c
B = b . b a    / c
```

- contains all items of *i* that have *b* as their control symb
- the point has been moved across *b*
- these are the productions that are worked on from here

Succ(*i*, *B*):

```
S = A B . c    / #
```

Succ(s, sym) is only the kernel of the new state; we still have to compute its closure

LALR(1) Table Generation

Extend the grammar by a pseudo production *S′* = *S* #
Create state 0 with the kernel S' = . S # // the only kernel with the point at the beginning

```
while (not all states visited) {
   s = next unvisited state;
   Compute the closure of s;
   for (all items of s) {
      switch (item kind) {
         case X = S . # :        generate accept; break;
         case X = α . y β / γ :  create new state s1 = Succ(s, y) if it does not exist yet;
                                 generate shift y, s1; break;
         case X = α . / γ :      generate reduce γ, (X = α); break;
         default:                generate error; break;
      }
   }
}
```

When checking for existing states only the kernels are considered without successor symbols, e.g.:

existing: E = v . / #+ + new: E = v . /)+ ⇒ E = v . /)#+

LALR(1) Table Generation

Grammar

0	S' = S #
1	S = A B
2	S = S A B
3	A = a
4	A = a a b
5	B = b
6	B = b b a

0	S' = . S #		shift	a	1
	S = . A B	/ #a	shift	A	2
	S = . S A B	/ #a	shift	S	3
	A = . a	/ b			
	A = . a a b	/ b			
1	A = a .	/ b	red	b	3
	A = a . a b	/ b	shift	a	4
2	S = A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	6
	B = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . A B	/ #a	shift	a	1 (!)
	A = . a	/ b	shift	A	7
	A = . a a b	/ b			
4	A = a a . b	/ b	shift	b	8
5	B = b .	/ #a	red	#,a	5
	B = b . b a	/ #a	shift	b	9
6	S = A B .	/ #a	red	#,a	1
7	S = S A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	10
	B = . b b a	/ #a			
8	A = a a b .	/ b	red	b	4
9	B = b b . a	/ #a	shift	a	11
10	S = S A B .	/ #a	red	#,a	2
11	B = b b a .	/ #a	red	#,a	6

Parser Table

0	S' = . S #		shift	a	1
	S = . A B	/ #a	shift	A	2
	S = . S A B	/ #a	shift	S	3
	A = . a	/ b			
	A = . a a b	/ b			
1	A = a .	/ b	red	b	3
	A = a . a b	/ b	shift	a	4
2	S = A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	6
	B = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . A B	/ #a	shift	a	1
	A = . a	/ b	shift	A	7
	A = . a a b	/ b			
4	A = a a . b	/ b	shift	b	8
5	B = b .	/ #a	red	#,a	5
	B = b . b a	/ #a	shift	b	9
6	S = A B .	/ #a	red	#,a	1
7	S = S A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	10
	B = . b b a	/ #a			
8	A = a a b .	/ b	red	b	4
9	B = b b . a	/ #a	shift	a	11
10	S = S A B .	/ #a	red	#,a	2
11	B = b b a .	/ #a	red	#,a	6

State transition table (parser table)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Parser Table as a List

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

error actions for NT symbols can never occur

	TS actions		*NTS actions*	
0	a	s1	S	s3
	*	error	A	s2
1	a	s4		
	*	r3		
2	b	s5	B	s6
	*	error		
3	a	s1	A	s7
	#	acc		
	*	error		
4	b	s8		
	*	error		
5	b	s9		
	*	r5		
6	*	r1		
7	b	s5	B	s10
	*	error		
8	*	r4		
9	a	s11		
	*	error		
10	*	r2		
11	*	r6		

- The actions in every state are checked sequentially.
- The last T action in every state is * *error* or * *ri* (lookahead tokens are not checked in *reduce* actions; if *reduce* was wrong this is detected at the next *shift*).
- lists are more compact but slower than tables.

LALR(1) Tables Versus LR(1) Tables

LR(1) is slightly more powerful than LALR(1), but its tables are about 10 times larger

LR(1) tables

- States are never merged
- The grammar is LR(1) if none of the states has one of the following conflicts:

shift-reduce conflict	shift a ... red a ...	the parser cannot decide with 1 symbol lookahead whether to shift or to reduce
reduce-reduce conflict	red a n red a m	the parser cannot decide with 1 symbol lookahead according to which production it should reduce

LALR(1) tables

- States can be merged if their kernels (without successor symbols) are identical

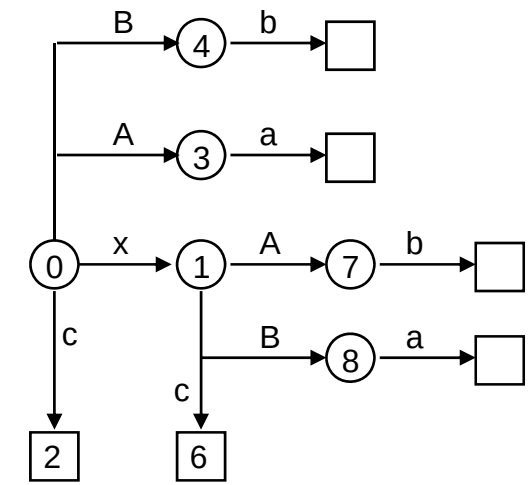
E = v . / #+ + E = v . /)+ ⇓ E = v . /)#+

- The grammar is LALR(1) if there is no reduce-reduce conflict after merging (shift-reduce conflicts cannot arise; why?)

Example: LR(1) Grammar which is not LALR(1)

Grammar

S' = S #
S = x A b
S = x B a
S = A a
S = B b
A = c
B = c



0	S' = . S #		shift	x	1
	S = . x A b	/ #	shift	c	2
	S = . x B a	/ #	shift	A	3
	S = . A a	/ #	shift	B	4
	S = . B b	/ #	shift	S	5
	A = . c	/ a			
	B = . c	/ b			
1	S = x . A b	/ #	shift	c	6
	S = x . B a	/ #	shift	A	7
	A = . c	/ b	shift	B	8
	B = . c	/ a			
2	A = c .	/ a	red	a	(A = c)
	B = c .	/ b	red	b	(B = c)
...	...				
6	A = c .	/ b	red	b	(A = c)
	B = c .	/ a	red	a	(B = c)
...	...				

Merging states 2 and 6 would lead to the following situation:

2	A = c .	/ ab	red	a,b	(A = c)
	B = c .	/ ab	red	a,b	(B = c)

reduce-reduce conflict!

SLR(1) Tables (Simple LR(1))

- SLR(1) parsing is less powerful than LALR(1) parsing
+ SLR(1) tables are simpler to generate than LALR(1) tables

SLR(1) items

shift item $X = \alpha \,.\, \beta$ ⟵ no successors are stored

reduce item $X = \alpha \,.\, / \gamma$ ⟵ γ = Follow(*X*), i.e. <u>all</u> successors of *X* in <u>any</u> context

Example: LALR(1) Grammar, which is not SLR(1)

Grammar

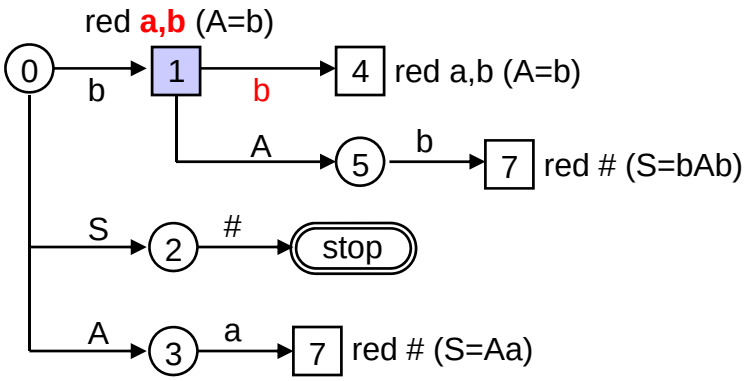
S' = S #
S = b A b
S = A a
A = b

0	S' = . S #	shift	b	1
	S = . b A b	shift	S	2
	S = . A a	shift	A	3
	A = . b			
1	S = b . A b	shift	b	4
	A = b .	red	a,b	(A = b)
	A = . b	shift	A	5

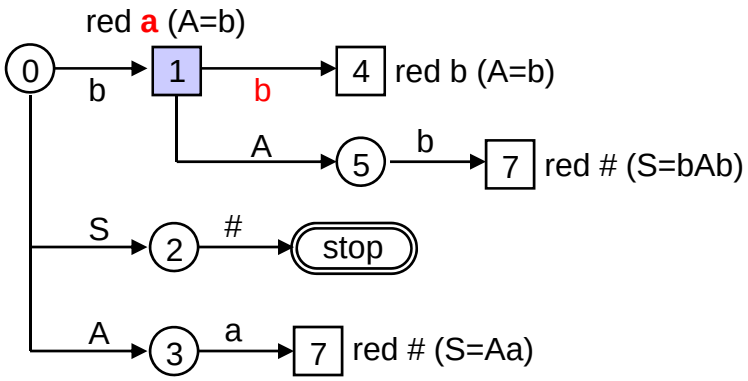
shift-reduce conflict, which would not arise in LALR(1) tables

Follow(S) = {#}
Follow(A) = {a, b}

SLR(1)



LALR(1)



7. Bottom-up Parsing

Back of the Envelope Calculation

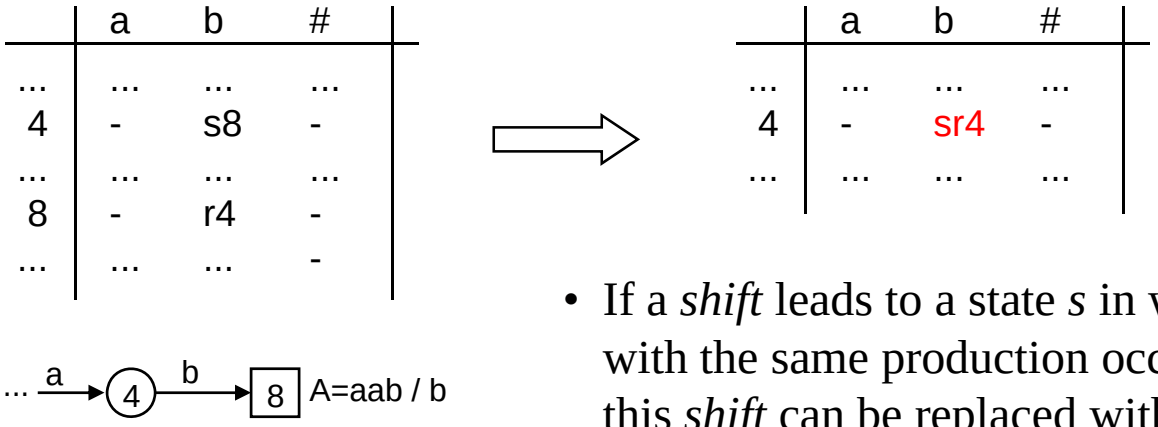
Assumption (e.g. C#)

80 terminal symbols
200 nonterminal symbols
2500 states ⇒ 280 x 2'500 = 700'000 entries

4 bytes / entry ⇒ 700'000 x 4 = 2'800'000 bytes = 2.8 MBytes

But the table size can be reduced by 90%

Combining Shift and Reduce Actions



	a	b	#
...	...	...	...
4	-	s8	-
...	...	...	...
8	-	r4	-
...	...	...	-

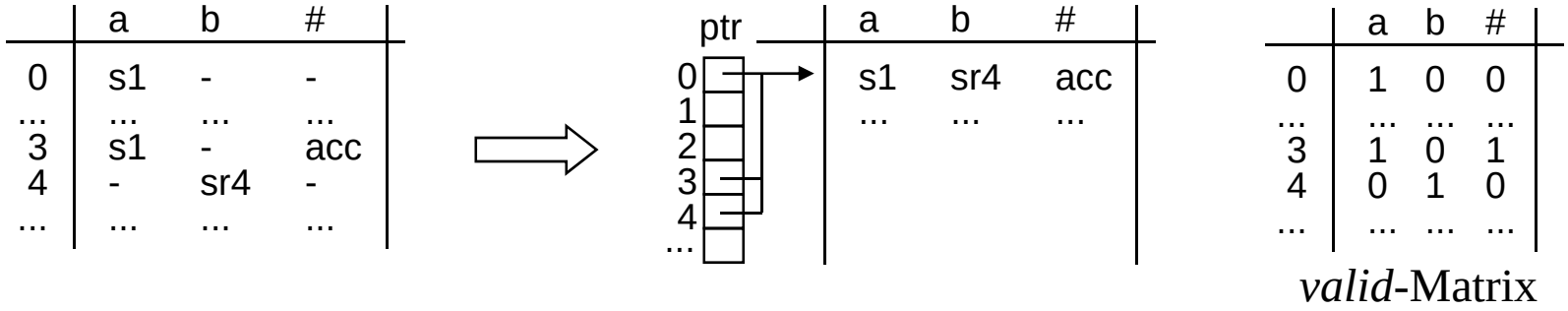
	a	b	#
...	...	...	...
4	-	sr4	-
...	...	...	...

- If a *shift* leads to a state *s* in which only *reduce* actions with the same production occur,
  this *shift* can be replaced with a *shift-reduce*.
- State *s* can be eliminated

Modification in the parsing algorithm

```
...
switch (op) {
   case shift:   // shift n
      state = n; sym = Next(); break;
   case shiftred:  // shiftred n
      for (int i = 0; i < length[n]-1; i++) Pop();
      a = action[Top(), leftSide[n]]; n = a % 256;  // shift n
      state = n;
      break;
   ...
```

Merging Rows



valid-Matrix

- Rows whose actions are not in conflict can be merged.
- Needs a pointer array in order to address the correct row indirectly.
- Needs a bit matrix that tells which actions are valid in every state.

- T and NT actions should be merged independently (more chances for merging).
- The same technique can also be used to merge columns.
- Further compaction is possible, but becomes more and more expensive.

Modifications in the parsing algorithm

```
BitArray[] valid;
short[] ptr;
...
a = action[ptr[state], sym];
op = a / 256; n = a % 256;
if (!valid[state][sym]) op = error;
switch (op) {
   ...
```

Example

Original table (6 columns x 12 rows x 2 bytes = **144** bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Combining shift and reduce (6 columns x 8 rows x 2 bytes = **96** bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Example (cont.)

After combining shift and reduce (**96** bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Merging rows (3 columns * 6 rows * 2 bytes
+ 2 * 8 rows * 2 bytes + 8 * 1 byte = 36 + 32 + 8 = **76** bytes)

T actions

	a	b	#
0,3,4	s1	sr4	acc
1	s4	r3	-
2,7,9	sr6	s5	-
5	r5	s9	r5

NT actions

	S	A	B
0,2	s3	s2	sr1
3,7	-	s7	sr2

valid

	a	b	#
0	1	0	0
1	1	1	0
2	0	1	0
3	1	0	1
4	0	1	0
5	1	1	1
7	0	1	0
9	1	0	0

7. Bottom-up Parsing

Semantic Actions

Are only allowed at the end of a production (i.e. when reducing)

Semantic actions in the middle of a production

```
X = a (. ... .) b.
```

must be implemented as follows:

```
X = a Y b.
Y = (. ... .).
```

← empty production; when it is reduced the semantic action is executed

Problem: this can destroy the LALR(1) property

```
X = a b c.
X = a (. ... .) b d.
```

CFG would be LALR(1) ⟹

```
X = a b c.
X = a Y b d.
Y = (. ... .).
```

Table generation

i					
	X = a . b c	/ #	shift	b	i+1
	X = a . Y b d	/ #	red	b	(Y=)
	Y = .	/ b	shift	Y	i+2

shift-reduce conflict

reason: the parser cannot work on both productio
in parallel any more, but must decide on one of t
two productions (i.e. whether to execute the
semantic action or not).

Attributes

- Every symbol has *one* output attribute (possibly an object with multiple fields)
- Whenever a symbol has been recognized its attribute is pushed on a *semantics stack*

Example

```
Expr<x> = Term<x>.
Expr<x> = Expr<x> "+" Term<y>        (. push(pop() + pop(); .).      // x = x + y;
Term<x> = Factor<x>.
Term<x> = Term<x> "*" Factor<y>      (. push(pop() * pop(); .).      // x = x * y;
Factor<x> = const<t>                 (. t = pop(); push(t.val); .).
Factor<x> = "(" Expr<x> ")".
```

Modifications in the Parser

Production table

	leftSide	length	sem
0	Expr	1	0
1	Expr	3	1
2	Term	1	0
3	Term	3	2
4	Factor	1	3
5	Factor	3	0

Semantics evaluator

```
void SemAction (int n) {
   switch (n) {
      case 1: Push(Pop() + Pop()); break;
      case 2: Push(Pop() * Pop()); break;
      case 3: Token t = Pop(); Push(t.val); break;
   }
}
```

Variables occurring in multiple semantic actions m
be declared global (problems with recursive NTS)

Parser

```
...
switch(op) {
   ...
   case reduce:  // red n
      if (sem[n] != 0) SemAction(sem[n]);
      for (int i = 0; i < length[n]; i++) Pop();
      ...
}
```

Input Attributes

Can be implemented as semantic actions with attribute assignments

```
X = a Y<❋v> b.
Y<❋w>  = ... .
```

can be written as

```
X = a (. w = v; .) Y b.
Y  = ... .
```

but this can destroy the LALR(1) property as we have seen

Therefore

- In practice LALR(1) parsers build a syntax tree (this can be implemented easily with output attributes and semantic actions at the end of productions).
- Semantic processing is then done on the syntax tree.

7. Bottom-up Parsing

Idea

Situation when an error occurs

$s_0 \ldots s_n \;.\; t_0 \ldots t_m \#$

stack input

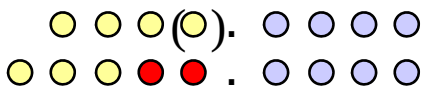
Goal: synchronize the stack and the input so that we have:

$s_0 \ldots s_i \;.\; t_j \ldots t_m \#$

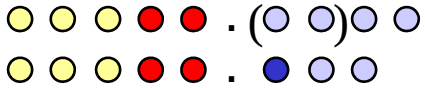
and t_j is accepted in s_i

Strategy

- Push or pop states



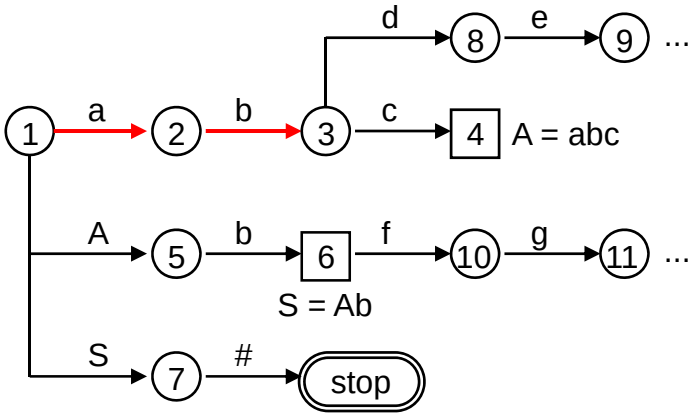
- Insert or delete tokens at the beginning of the input stream



Algorithm

1. **Search escape route**
   - Replace t_0 .. t_m with a virtual input v_0 .. v_k, which drives the parser from the error state s_n into a final state as quickly as possible.
   - While parsing v_0 .. v_k collect all tokens which are valid in the traversed states ⓿ anchors
2. **Delete invalid tokens**
   - Skip tokens from the input t_0 .. t_m, until a token t_j occurs, which is an anchor.
3. **Insert missing tokens**
   - Drive the parser from s_n with v_0 .. v_k until it gets into a state s_i in which t_j is accepted.
   - Insert all "read" virtual tokens from v_0 .. v_k into the input in front of t_j.

Example



input: a b b #

Search escape route

virtual input: c b #

anchors:

3	c, d
5	b
6	f
7	#

Insert missing tokens

insert *c*, in order to get to 4 and 5.

In 5 the parser can continue with *b*.

Delete missing tokens

Nothing is skipped because the next input token *b* is already an anchor

Corrected input

a b c b #

Error Messages

They simply report what was inserted or deleted

- If the tokens *a*, *b*, *c* were deleted from the input:

```
line ... col ... :  "a b c" deleted
```

- If the tokens *x*, *y* were inserted into the input:

```
line ... col ... :  "x y" inserted
```

- If the tokens *a*, *b*, *c* were deleted and the tokens *x*, *y* were inserted:

```
line ... col ... :  "a b c" replaced by "x y"
```

Another Example (With a Parser Simulation)

grammar

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

erroneous input

a a a b #

Every state has a "guide symbol". We will explain later how this is computed.

Start of parsing

stack	*input*	*action*
0	a a a b #	s1
0 1	a a b #	s4
0 1 4	a b #	-- error!

Search escape route and collect anchors

stack	*guide*	*action*	*anchors*
0 1 4	b	s8	b
0 1 4 8	b	r4, s2	b
0 2	b	s5	b
0 2 5	a	r5, s6	a, b, #
0 2 6	a	r1, s3	a, #
0 3	#	acc	a, #

anchors = {*a*, *b*, #}

Example (cont.)

grammar

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

remaining input

a b #

anchors

{a, b, #}

Skip input

Nothing is skipped because the next input token *a*
is already in the anchor set {*a*, *b*, #}

Insert missing tokens

stack	*guide*	*action*	*inserted*
0 1 4	b	s8	b
0 1 4 8	b	r4, s2	
0 2	b	s5	b
0 2 5			

← only *shift* causes a token to be inserted, not *reduce*

Here we can continue with *a*

Example (cont.)

grammar

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

remaining input

a b #

Parsing continues

stack	*input*	*action*
0 2 5	a b #	r5, s6
0 2 6	a b #	r1, s3
0 3	a b #	s1
0 3 1	b #	r3, s7
0 3 7	b #	s5
0 3 7 5	#	r5, s10
0 3 7 10	#	r2, s3
0 3	#	acc

Corrected input

a a b b a b #

Error message

line ... col ...: "b b" inserted

Computing the Guide Symbols?

1. Order the productions so that the first production of every NTS is non-recursive and as short as possible

```
S = A B.
S = S A B.   ← the recursive production becomes the second one
A = a.
A = a a b.   ← the longer production becomes the second one
B = b.
B = b b a.   ← the longer production becomes the second one
```

2. Compute the LALR(1) tables in the normal way.
   The first generated T action in every state determines the guide symbol for this state

```
0   S' = . S #              shift   a    1   ← guide symbol is a
    S = . A B      / #a     shift   A    2
    S = . S A B    / #a     shift   S    3
    A = . a        / b
    A = . a a b    / b
```

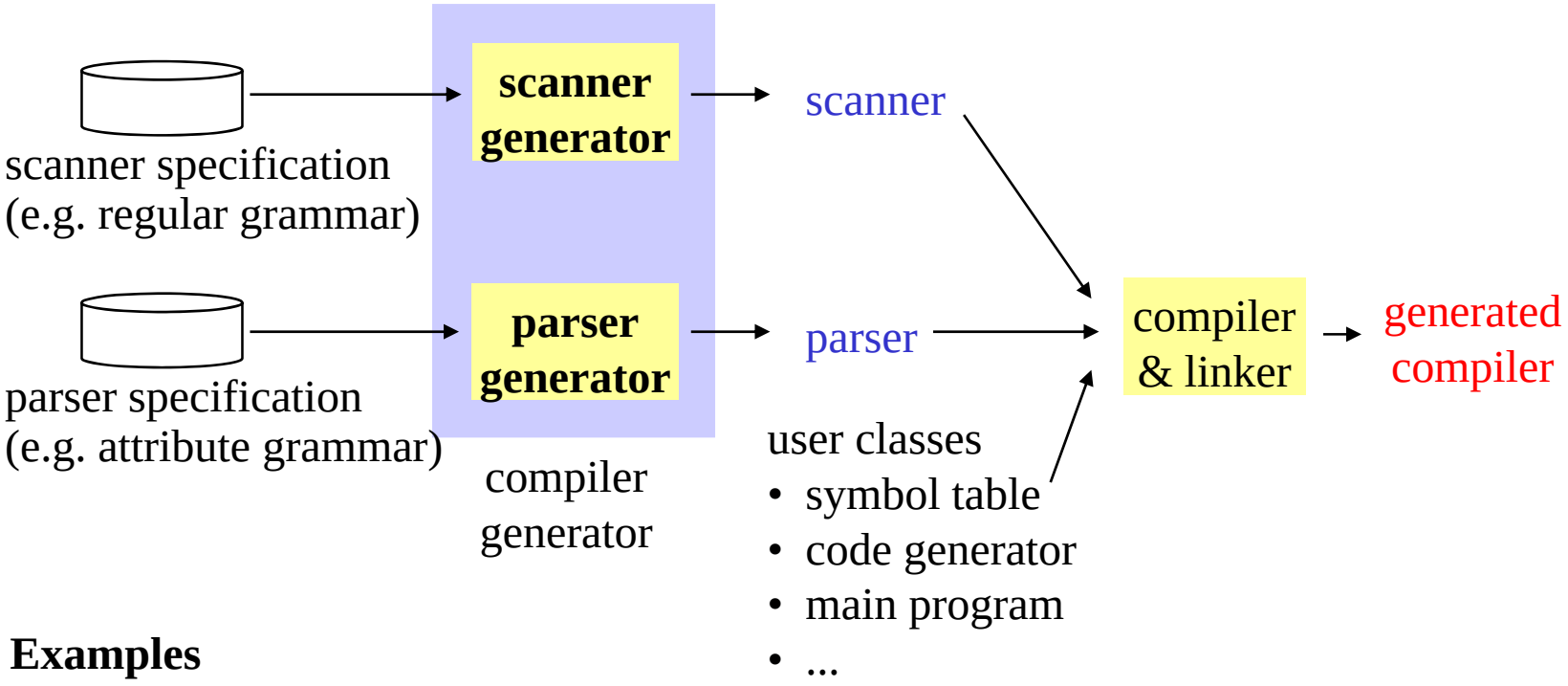
Assessment of this Error Handling Technique

- Does not slow down error-free parsing
- Always terminates (# is always an anchor)
- Does not only *report* errors but even "*corrects*" them (but the correction is not always what we want).
- Generates good error messages

8. Compiler Generators

How Compiler Generators Work

They generate parts of a compiler from a concise specification
(generated parts e.g.: scanner, parser, code generator, tree optimizer, ...)



Examples

Yacc	parser generator for C and Java
Lex	scanner generator for C, Java and C#
Coco/R	scanner and parser generator for Java, C#, Modula-2, Oberon, ...
...	

8. Compiler Generators

Yacc - Yet another compiler compiler

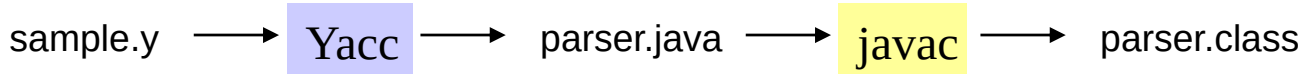
History

1975 developed at **Bell Labs** (together with C and Unix)

- generates LALR(1) parsers
- originally under Unix, today also under Windows, Linux
- originally for C, today also for Java

Use

we describe the Java version here



Major Java versions today

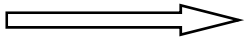
Bison	GNU version of Yacc http://www.gnu.org/software/bison/bison.html
Byacc	Berkeley Yacc http://byaccj.sourceforge.net/

Yacc Input Language

General format

```
    %{  Java package and import lines %}
    Yacc declarations
    (tokens, operator precedence rules, ...)
%%
    productions
%%
    Java declarations (fields, methods)
```

translated into



```
class parser {
    ...
    public void yyparse() {
        ... parser ...
    }
}
```

Yacc — Productions and Sem. Actions

Productions

```
Grammar    = {Production}.
Production = NT ":" Alternative {"|" Alternative} ";"
Alternative = {NT | T} [SemAction].
SemAction  = "{" ... arbitrary Java statements ... "}".
NT         = ident.
T          = ident | charConst.
```

Example

```
expr:  term              { $$ = $1; }
     | expr '+' term  { $$.ival = $1.ival + $3.ival; } ;
```

Semantic Actions

- may contain arbitrary Java statements
- may only occur at the end of an alternative
- attributes are denoted by special names:

  `$$` attribute of the left-hand side NTS

  `$i` attribute of the right-hand side symbol *i*

  (`$1` = attr. of first symbol, `$2` = attr. of second symbol, ...)

Yacc — Attributes

For terminal symbols

- Are delivered by the scanner (scanner is either hand-written or generated with *Lex*)
- Every token has an attribute of type *parserval*

```
class parserval {
    int ival;          // token value if the token should have an int attribute
    double dval;       // token value if the token should have a double attribute
    String sval;       // token value, e.g. for ident and string
    Object obj;        // token value for more complex tokens
    parserval(int val) {...}          // constructors
    parserval(double val) {...}
    parserval(String val) {...}
    parserval(Obj val) {...}
}
```

- The scanner returns the attributes in the global variable *yylval*

scanner

```
yylval = new parserval(n);
```

access in a sem. action

```
{ ... $1.ival ... }
```

For nonterminal symbols

- Every NTS has an attribute `$$` of type *parserval* (more complex values stored in `$$.obj`)
- Every assignment to `$$` pushes the NTS attribute onto an attribute stack
  Accesses to `$1`, `$2` pop the attribute from the stack

Yacc — Java Variables and Methods

Are declared after the second %%

```
    %{ ... imports ... %}
    ... tokens ...
%%
    ... productions ...
%%
    ... Java declarations ...
```

- become fields and methods of the parser
- can be accessed in semantic actions

At least the following methods must be implemented in the Java declarations

```
void yyerror(String msg) {...}
```

for printing error messages

```
int yylex() {...}
```

scanner (yields token numbers and fills *yylval*)

```
public static void main(String[] arg) {
    ... initializations for yylex ...
    yyparse();
}
```

main program

Example: Compiler for Arithmetic Expressions


```
/* declaration of all tokens which are not strings */
%token number

%%

/* productions: first NTS is the start symbol */
input: expr             { System.out.println($1.ival); } ;

expr: term              { $$ = $1; }
    | expr '+' term     { $$.ival = $1.ival + $3.ival; } ;

term: factor            { $$ = $1; }
    | term '*' factor   { $$.ival = $1.ival * $3.ival; } ;

factor: number          { $$ = $1; }
    | '(' expr ')'      { $$ = $2; } ;

%%

int yylex() {...}
void yyerror(string msg) {...}
public static void main(String[] arg) {...}
```

Token numbering conventions

- *eof* == 0
- token number of characters is their Ascii value (e.g. '+' == 43)
- YYERRCODE == 256
- other tokens are numbered consecutively starting with 257 (e.g. *number* == 257);
  they can be accessed in productions and in *yylex()* using their declared name.

Yacc — Operator Precedence

The following grammar actually already specifies the operator precedence

```
expr:   term | expr '+' term ;
term:   factor | term '*' factor ;
factor: number | '(' expr ')' ;
```

- '*' has precedence over '+'
- operators are left-associative: a*b*c == (a*b)*c

Instead, one can also write the following in Yacc:

```
%token  number
%left   '+'
%left   '*'

%%

input: expr             { System.out.println($1.ival); } ;

expr: number            { $$ = $1; }
    | expr '+' expr     { $$.ival = $1.ival + $3.ival; }
    | expr '*' expr     { $$.ival = $1.ival * $3.ival; }
    | '(' expr ')'      { $$ = $2; }

%%
...
```

- *%left* the operator is left-associative
  a+b+c == (a+b)+c
- operators are declared in ascending order of priority:
  '*' has precedence over '+'
- this grammar does not specify any operator precedence
- the precedence is rather specified by *%left* or *%right*

Yacc — Error Handling

***error* alternatives**

For certain NTS (e.g. *Statement*, *Expression*, ...) the user must specify *error* alternatives

```
A:  ...
  |  ...
  |  error α {...} ;
```

α ... arbitrary sequence of T and NT symbols

Meaning: If there is an error in *A* the parser does the following:

- it pops states from the stack until it gets to a state in which a shift action with the *error* token is valid
- *shift error*
- it skips input tokens until it detects a token sequence which can be reduced to α (the stack end now contains: *error* α)
- it reduces *error* α to *A* and executes the corresponding semantic action

Example

```
Statement = ...
  | error ';' ;
```

skips everything up to the next ';'

8. Compiler Generators

Lex — Scanner Generator

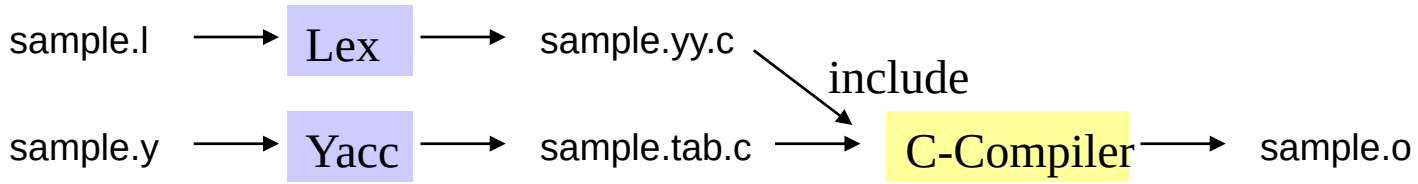
History

1975 developed at **Bell Labs**

- generates a scanner in form of a DFA
- originally a Unix tool, today also for Windows
- originally for C, today also for Java
- usually cooperates with *Yacc*

Use

we describe the C version here



Major versions today

flex	GNU version of Lex (for C) http://www.gnu.org/software/flex/
JLex	Java version with slightly different input syntax; incompatible with Bison or Byacc http://www.cs.princeton.edu/~appel/modern/java/JLex/
CsLex	C# version, derived from JLex http://www.cybercom.net/~zbrad/DotNet/Lex

Example of a Lex Description


```
%{ ... e.g. include directives for token numbers exported by the parser ... %}

/* macros */
delim       [  \t\n]                /* blank, tab, eol */
ws          {delim}+                /* {...} ... use of a macro */
letter      [A-Za-z]
digit       [0-9]
id          {letter} ({letter} | {digit})*
number      {digit}+

%% /* token declarations described as regular expressions */
{ws}        {}                      /* no action */
if          { return IF; }          /* constants like IF are imported from the parser */
then        { return THEN;}
else        { return ELSE; }
{id}        { yylval = storeId(yytext, yyleng); return ID; }
{number}    { yylval = convert(yytext, yyleng); return number; }
<           { return yytext[0]; }
>           { return yytext[0]; }
.           {}                      /* . denotes any character */

%% /* semantic routines */
int storeId(char* text, int len) {...}
int convert(char* text, int len) {...}
```

Generated Scanner

The scanner specification is translated into a function

```
int yylex() {...}
```

which is included into the parser

yylex() also returns the token attributes as global variables

```
int yylval;      /* attribute if the token has a numeric value */
char* yytext;    /* token text (attribute of ident, string, ...) */
int yyleng;      /* lengh of the token text */
int yylineno;    /* line number of the token */
```

The parser declares (and exports) token codes

```
%token IF
%token THEN
...
```

Regular Expressions in Lex

Elements of regular expressions

abc	the string "abc"; every character except ()[]{}*+?|^$.\ denotes itself
.	any character except \n (eol)
x*	0 or more repetitions of x
x+	1 or more repetitions of x
x?	0 or 1 occurrence of x (optional occurrence)
(...|...)	for grouping of alternatives
[...]	set of all characters between the brackets (e.g. [A-Za-z0-9$])
{...}	use of a macro
^	line start
$	line end
\udddd	character in Unicode

Conventions

- the scanner recognizes the token with the longest possible character sequence (e.g. *iff* is recognized as *ID* and not as *if*)
- the scanner tries to match the token declarations in sequential order (tokens declared first have priority over tokens declared later)

8. Compiler Generators

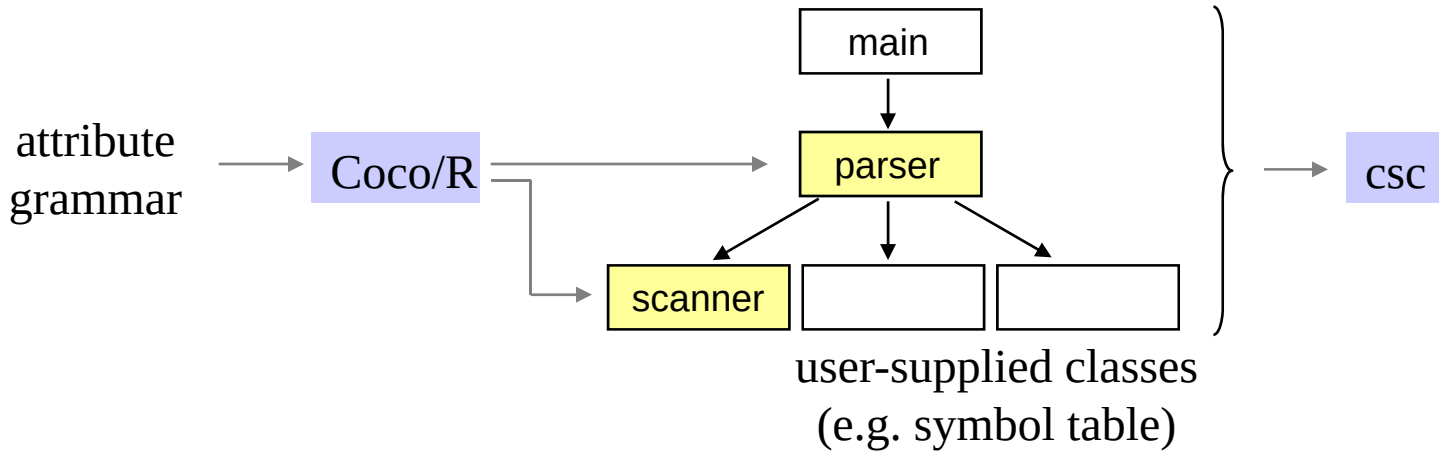
Coco/R - Compiler Compiler / Recursive Descent

History

1980 developed at the university of Linz (Rechenberg, Mössenböck)

- generates a scanner and a parser from an attribute grammar
  - scanner as a DFA
  - recursive descent parser
- there are versions for C#, Java, C/C++, Delphi, Modula-2, Oberon, Python, ...
- published under GPL: http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

Use



Example: Compiler for Arithmetic Expressions


```
COMPILER Calc                 /* grammar name = start symbol */
CHARACTERS                    /* character sets used in token declarations */
   digit = '0' .. '9'.
   tab = '\t'. cr = '\r'. lf = '\n'.
TOKENS                        /* declaration of all tokens which are not literals */
   number = digit {digit}.
COMMENTS                      /* declaration of comments */
   FROM "//" TO cr lf
   FROM "/*" TO "*/" NESTED
IGNORE tab cr lf              /* these characters are ignored as white space */
PRODUCTIONS
   Calc                       (. int x; .)
   = "CALC" Expr<out x>       (. System.Console.WriteLine(x); .) .

   Expr<out int x>            (. int y; .)
   =  Term<out x>
      { '+' Term<out y>       (. x = x + y; .)
      }.

   Term<out int x>            (. int y; .)
   =  Factor<out x>
      { '*' Factor<out y>     (. x = x * y; .)
      }.

   Factor<out int x>
   =  number                  (. x = Convert.ToInt32(t.val); .)
   |  '(' Expr<out x> ')'.
END Calc.
```

Coco/R — Attributes

Terminal symbols

- terminal symbols have no explicit attributes
- their values can be accessed in sem. actions using the following variables declared in the

Token **t**; the most recently recognized token

Token **la**; the lookahead token (not yet recognized)

Example

```
Factor<out int x> = number    (. x = Convert.ToInt32(t.val); .)
```

```
class Token {
   int kind;       // token code
   string val;     // token value
   int pos;        // token position in the source text (starting at 0)
   int line;       // token line (starting at 1)
   int col;        // token column (starting at 0)
}
```

Nonterminal symbols

- NTS can have any number of input attributes

  formal attr.: `A<int x, char c> = ... .` actual attr.: `... A<y, 'a'> ...`

- NTS can have any number of output attributes

  `B<out int x, out int y> = ... .` `... B<out z, out n> ...`

Coco/R — Semantic Processing

Semantic Actions

- arbitrary Java code between (. and .)
- can occur anywhere in productions
- on the left-hand side of a production they are regarded as declarations

```
Term<out int x>              (. int y; .)
=  Factor<out x>
   { '*' Factor<out y>       (. x = x * y; .)
   }.
```

declaration

semantic action

Semantic Declarations

- occur at the beginning of a compiler specification
- are used to declare arbitrary fields and methods of the parser
- imports can also be specified

```
COMPILER Sample
   using System.Collections;
   static IList myList;
   static void AddToList (int x) {...}
CHARACTERS
   ...
```

Of course, semantic actions can also access fields and methods of classes other than the parser.

Coco/R - Parsing Methods

Every production is translated into a parsing method

```
Expr<out int x>                (. int y; .)
=  Term<out x>
   { '+' Term<out y>           (. x += y; .)
   }.
```

becomes

```
static void Expr (out int x) {
   int y;
   Term(out x);
   while (la.kind == plus) {
      Scan();
      Term(out y);
      x += y;
   }
}
```

Coco/R - Syntax Error Handling

The parser uses the *special anchor* technique

Synchronization points

must be marked by SYNC

```
Statement
= SYNC
(   Assignment
|   IfSatement
|   ...
).
```

if *la.kind* ∉ Follow(SYNC)
an error is reported and tokens are skipped
until *la.kind* ∈ Follow(SYNC) ⇧ {*eof*}

Spurious error messages are suppressed if less than 3 tokens have been recognized since the last error.

Weak separators

Separators at the beginning of an iteration can be marked as WEAK

```
FormalPars
= "(" Param
   {  WEAK ','
      Param
   } ')'.
```

If the separator is missing or mistyped, the loop is not terminated prematurely, but the parser synchronizes with First(*Param*) ⇧ Follow({...}) ⇧ {*eof*}

Coco/R — Grammar Tests

LL(1) test

```
A = a [B] C d
  | B a.
B = a b.
C = a [d].
```

Coco/R prints the following warnings

LL1 warning in A: a is start & successor of deletable structure
LL1 warning in A: a is start of several alternatives
LL1 warning in C: d is start & successor of deletable structure

Completeness test

Is there a production for every NTS?

Non-redundancy test

Does the grammar contain productions which can never be reached?

Derivability test

Can every NTS be derived into a string of terminal symbols?

Non-circularity test

Are there NTS which can be derived (directly or indirectly) into themselves?

Coco/R — Pragmas

Pragmas are terminal symbols

- which can occur anywhere in the input
- which are not part of the syntax
- which must be processed semantically

e.g. compiler options

```
COMPILER X
CHARACTERS ...
TOKENS ...
PRAGMAS
   PrintOption = "$print".      (. option[print] = true; .)
   DbgOption  = "$debug".     (. option[debug] = true; .)
...
```

Whenever the string

```
$print
```

occurs in the input text the semantic action

```
option[print] = true;
```

is executed

Coco/R — The ANY Symbol

In the declaration of character sets

it describes complementary character sets

```
CHARACTERS
   letter = 'A' .. 'Z' + 'a' .. 'z'.
   noLetter = ANY - letter.
...
```

all characters which are not letters

In productions

it describes any tokens which cannot be matched by other alternatives

```
PlaceHolder
= ident
|  ANY.
```

any token which is not *ident* or *eof*

```
SemAction = "(." { ANY } ".)".
```

any token which is not ".)" or *eof*

Coco/R — LL(1) Conflict Resolution

Conflict resolution by a multi-symbol lookahead

```
Statement
=  IF (IsAssignment())
   Designator "=" Expr ";"
|  Designator "(" ActualParams ")" ";"
|  ... .
```

```
static boolean IsAssignment () {
   Token x = la;
   while (x.kind != _assign && x.kind != _lpar)
      x = Scanner.Peek();
   return x.kind == _assign;
}
```

Scanner.Peek() ... reads ahead without removi
tokens from the input stream

Token names (*_assign*, *_lpar*, ...) are generated from the TOKENS sections

Conflict resolution by a semantic check

```
Factor
=  IF (IsCast())
   '(' ident ')' Factor    /* type cast */
|  '(' Expr ')'            /* nested expression */
| ... .
```

```
static boolean IsCast () {
   Token x = Scanner.Peek();
   if (x.kind == _ident) {
      Symbol s = Tab.Find(x.val);
      return s.kind == Symbol.Kinds.Type;
   } else return false;
}
```

Coco/R — Frame Files

The scanner and the parser are generated from frame files (ordinary text files)

e.g. ***Scanner.frame***

```
public class Scanner {
   const char EOL = '\n';
   const int  eofSym = 0;
-->declarations
...
   static Token NextToken () {
      while (ignore[ch]) NextCh();
-->scan1
      t = new Token();
      t.pos = pos; t.col = pos - lineStart + 1; t.line = line;
      int state = start[ch];
      StringBuilder buf = new StringBuilder(16);
-->scan2
      ...
   }
   ...
}
```

Coco/R inserts code
at these positions

By modifying the frame files
the scanner and the parser can be
adapted to user needs
(to a certain degree)

Coco/R — Interfaces

Scanner

```
public class Scanner {
   public static void    Init (string sourceFileName) {...}
   public static void    Init (Stream s) {...}
   public static Token   Scan () {...}
   public static Token   Peek () {...}
   public static void    ResetPeek () {...}
}
```

Parser

```
public class Parser {
   public static Token   t;
   public static Token   la;
   public static void    Parse () {...}
   public static void    SemErr (string msg) {...}
}
```

Error message class

```
public class Errors {
   public static int     count = 0;
   public static string  errMsgFormat = "-- line {0} col {1}: {2}";
   public static void    SynErr (int line, int col, int n);
   public static void    SemErr (int line, int col, int n);
   public static void    Error (int line, int col, string msg);
   public static void    Exception (string msg);
}
```