Programming Languages and Compilers

Context-free Grammars and Syntactic Analysis

Dr. Peter Pfahler
Based on the lecture by Prof. Dr. Uwe Kastens

Universität Paderborn
Fakultät EIM
Institut für Informatik

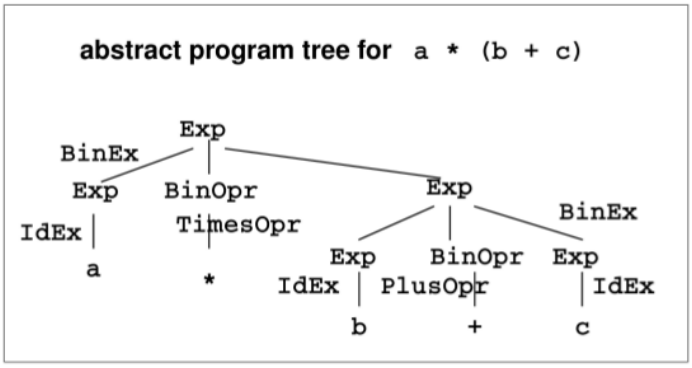
Winter 2016/2017

Syntactic Analysis

Input: Token Sequence

Tasks	**Compiler Module**
Read token sequence	Interface to lexical analysis
Construct a derivation according to the concrete syntax	Parser, central phase, stack automaton
Build a structure tree according to the abstract syntax	Tree construction
Detect and report errors	Error handling

Output: *Abstract Syntax Tree (AST)* , a condensed version of the derivation tree:



The terms

- Abstract Syntax Tree
- Abstract Program Tree
- Abstract Structure Tree

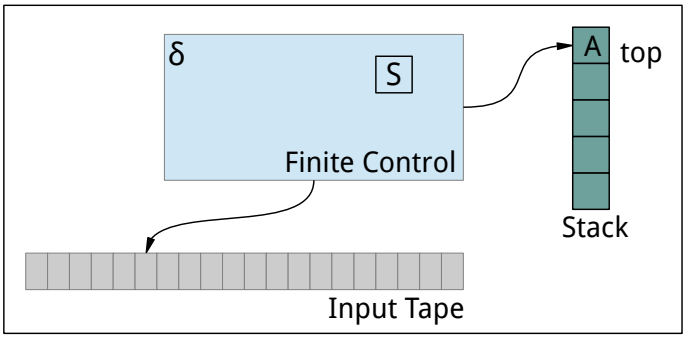
are used synonymously.

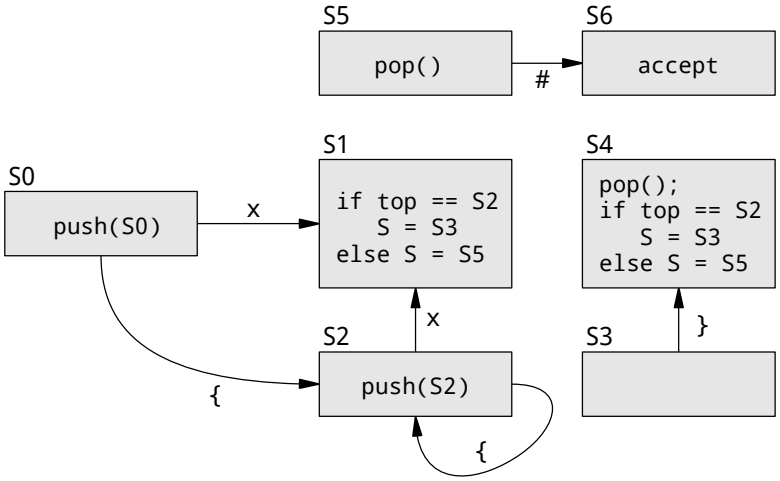
Stack Automata

Formal model to recognize context-free languages.

Example "Nested Blocks":

```
S ::= block
block ::= 'x'
block ::= '{' block '}'
```





Transitions depending on both input and top-of-stack.

State transitions can manipulate the stack.

Section Structure

The section *Context-free Grammars and Syntactic Analysis* will be structured as follows:

1. Grammar Design
   - Concrete and Abstract Grammars
   - Expression Grammars
   - A Strategy for Grammar Development
   - Ambiguity and Unbounded Lookahead
2. Parsing Methods: Top-Down vs. Bottom-Up Parsing
3. Top-Down Parsing
   - Recursive Descent Parsers
   - Grammar Transformations for LL(1), Handling EBNF
4. Bottom-Up Parsing
   - Shift-Reduce Parsers
   - LR(0) and LR(1)-Parser Construction
   - Hierarchy of Grammar Classes
   - Implementing LR-automata
5. Syntax Error Handling
6. Parser Generators

Concrete and Abstract Syntax

Concrete Syntax

- context-free grammar
- defines the source structure
- unambiguous
- specifies parser construction and derivation

Abstract Syntax

- context-free grammar
- defines the abstract syntax trees
- usually ambiguous
- semantic analysis and transformation is based on it

Actions added to the concrete grammar specify abstract syntax tree construction:

```
Expr ::= Expr AddOpr Term &'MkNode(BinExpr, ...);'
```

The abstract syntax ommits

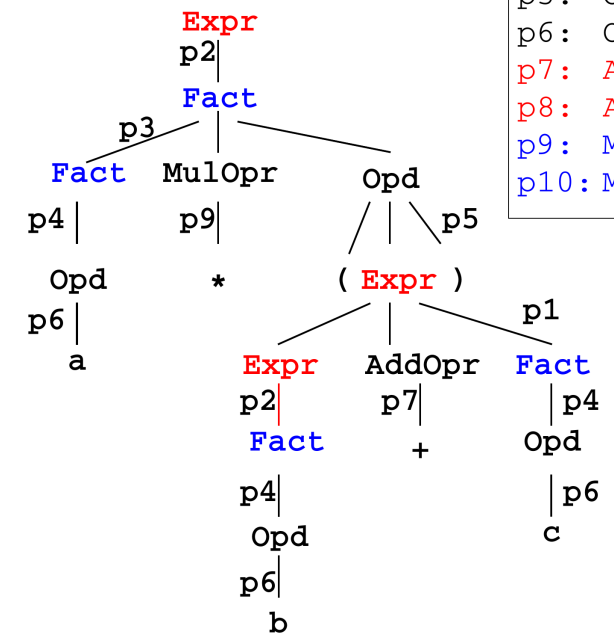
- Chain productions having only syntactic purpose
- Terminal symbols which are not relevant semantically

The abstract syntax can be generated from the concrete syntax *and* a symbol mapping, like e. g.: `Exp = {Expr, Term, Fact}` .

Example: Concrete Expression Grammar

name	production			*action*
p1:	Expr	::=	Expr AddOpr Fact	*BinEx*
p2:	Expr	::=	Fact	
p3:	Fact	::=	Fact MulOpr Opd	*BinEx*
p4:	Fact	::=	Opd	
p5:	Opd	::=	'(' Expr ')'	
p6:	Opd	::=	Ident	*IdEx*
p7:	AddOpr	::=	'+'	*PlusOpr*
p8:	AddOpr	::=	'-'	*MinusOpr*
p9:	MulOpr	::=	'*'	*TimesOpr*
p10:	MulOpr	::=	'/'	*DivOpr*

derivation tree for `a * (b + c)`

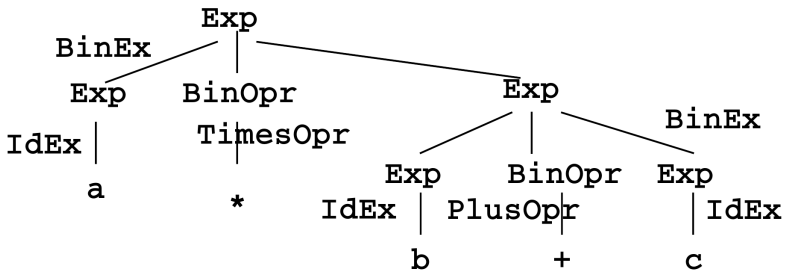


`+, -` lower precedence
`*, /` higher precedence

Example: Abstract Expression Grammar

name	production
BinEx:	Exp ::= Exp BinOpr Exp
IdEx:	Exp ::= Ident
PlusOpr:	BinOpr ::= '+'
MinusOpr:	BinOpr ::= '-'
TimesOpr:	BinOpr ::= '*'
DivOpr:	BinOpr ::= '/'

abstract program tree for `a * (b + c)`



symbol classes: `Exp = { Expr, Fact, Opd }`
`BinOpr = { AddOpr, MulOpr }`

Actions of the concrete syntax: **productions** of the abstract syntax to create tree nodes for
no action at a concrete chain production: **no tree node** is created

A Strategy for Grammar Development

1. **Examples**: Write at least one example for every intended language construct. Keep the examples for checking the grammar and the parser.
2. **Sub-grammars**: Decompose a non-trivial task into topics covered by sub-gammars, e.g. statements, declarations, expressions, over-all structure.
3. **Top-down**: Begin with the start symbol of the (sub-)grammar, and refine each nonterminal according to steps 4 - 7 until all nonterminals of the (sub-)grammar are refined.
4. **Alternatives**: Check whether the language construct represented by the current nonterminal, say Statement, shall occur in structurally different alternatives, e.g. while-statement, if-statement, assignment. Either introduce chain productions, like
   `Statement ::= WhileStatement | IfStatement | Assignment.`
   or apply steps 5 - 7 for each alternative separately.
5. **Consists of**: For each (alternative of a) nonterminal representing a language construct explain its immediate structure in words, e.g. „A Block is a declaration sequence followed by a statement sequence, both enclosed in curly braces.“ Refine only one structural level. Translate the description into a production. If a sub-structure is not yet specified introduce a new nonterminal with a speaking name for it, e.g.
   `Block ::= '{' DeclarationSeq StatementSeq '}'.`
6. **Natural structure**: Make sure that step 5 yields a „natural“ structure, which supports notions used for static or dynamic semantics, e.g. a range for valid bindings.
7. **Useful patterns**: In step 5 apply patterns for description of sequences, expressions, etc.

Patterns for Sequences

Description	Left-Recursion	Right-Recursion
Non-empty Sequence	`A ::= A b` `A ::= b`	`A ::= b A` `A ::= b`
Possibly empty Sequence	`A ::= A b` `A ::=`	`A ::= b A` `A ::=`
Non-empty separated Sequence	`A ::= A s b` `A ::= b`	`A ::= b s A` `A ::= b`

Possibly empty separated Sequence	`A ::= B  A ::=`
	`B ::= B s b`
	`B ::= b`
		`A ::= B  A ::=`
		`B ::= b s B`
		`B ::= b`

Example: A formal parameter list

```
formparams ::= fparams
formparams ::=
fparams    ::= fparam
fparams    ::= fparams ',' fparam
fparam     ::= type Identififer
```


Patterns for Expression Grammars

Expression grammars are **systematically** constructed, such that **structural properties** of expressions are defined:

one level of precedence, **binary** operator,left-associative:

```
A ::= A Opr B
A ::= B
```

one level of precedence, **binary** operator,right-associative:

```
A ::= B Opr A
A ::= B
```

one level of precedence, **unary Operator**, prefix:

```
A ::= Opr A
A ::= B
```

one level of precedence, **unary Operator**, postfix:

```
A ::= A Opr
A ::= B
```

Elementary operands: **only derived** from the nonterminal of the **highest precedence** level (be H here):

```
H ::= Ident
```

Expressions in parentheses: only derived from the nonterminal of the **highest precedence** level (assumed to be H here); **contain** the nonterminal of the **lowest precedence level** (be A here):

```
H ::= '(' A ')'
```

Grammar Design

Read grammars before writing a new grammar.

Apply grammar patterns systematically:

- repetitions
- optional constructs
- precedence, associativity of operators

Syntactic structure should reflect semantic structure.
Example: A range in the sense of scope rules should be represented by a single subtree of the abstract structure tree.
Difficult, if the syntax does not reflect this, e.g. in Pascal:

```
funDecl ::= funHead block
funHead ::= 'function' identifier formParams ':' resultType ';'
```

`formParams` together with `block` form a range. The function name (`identifier`) does not belong to that range, but to the enclosing one.

Syntactic Restrictions versus Semantic Conditions

Language constraints should not be handled syntactically if:

- Restriction can not be decided syntactically, e.g. type check in expressions:

```
BoolExpression ::= IntExpression '<' IntExpression
```

- Restriction can not always be decided syntactically, e. g. disallow array type to be used as function result:

```
Type ::= ArrayType | NonArrayType | Identifier
ResultType ::= NonArrayType
```

  If a type identifier may specify an array type, a semantic condition is needed, anyhow.

- Syntactic restriction is unreasonably complex, e. g. distinction of expressions with values known at compile-time from ordinary expressions requires duplication of the expression syntax.

Eliminate Ambiguities

by uniting syntactic constructs and distinguishing them semantically:

- Java:

```
ClassOrInterfaceType ::= ClassType | InterfaceType
ClassType            ::= TypeName
InterfaceType        ::= TypeName
```

  ⇒ Replace first production by `ClassOrInterfaceType ::= TypeName`
  Semantic analysis distinguishes between class type and interface type

- Pascal:

```
factor ::= variable | ... | functionDesignator
variable::= entireVariable | ...
entireVariable::= variableId
variableId::= ident                    (**)
functionDesignator ::= functionId      (*)
functionDesignator ::= functionId '(' actParams ')'
functionId ::= ident
```

  ⇒ Eliminate alternative marked (*). Semantic analysis checks whether (**) is a function identifier