Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 2

Outline

Software Technology Systems

Outline

Parse tree

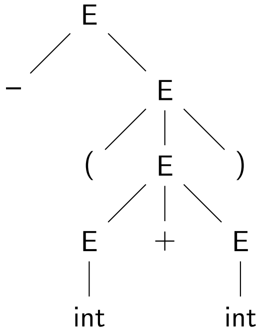
A parse tree is a graphical representation of a derivation sequence.

- The leaves of a parse tree are terminal symbols, the inner nodes are non-terminals.
- The parse tree is constructed inductively: the left-hand side of a production corresponds to a node, its right hand-side to its children.
- The leaves of a parse tree, in-place concatenated, form a legal sentence.

Example (parse tree construction)

Let E be an abbreviation for “expr.” Consider the derivation and the corresponding parse tree (assume the previous grammar):

$$E \rightarrow \text{-}E \rightarrow \text{-}(E) \rightarrow \text{-}(E + E) \rightarrow \text{-}(int + E) \rightarrow \text{-}(int + int)$$



In-class exercise (parse trees)

Grammar:

- *expr* $\rightarrow$ *expr op expr* | (*expr*) | -exp | int
- *op* $\rightarrow$ + | * | - | /

Ambiguities

An ambiguous grammar is a grammar that produces two different parse trees for at least one sentence.

Since certain types of *parsers* require unambiguous grammars, one often tries to make a grammar unambiguous. The two major ways are:

- Refactoring or transforming the grammar (if possible)
- Adding meta-rules, e.g., for precedences

Precedence & associativity

Precedence and associativity are not directly expressible in a grammar. They are defined in rules outside the grammar.

Motivation

- Well-definedness: in parsing, the result (parse tree) is well-defined.
- Modeling a domain faithfully: in mathematics, e.g., * has higher precedence than $+$, and * is left associative
- Convenience: save writing parentheses. Ex.: in functional programming, function application has higher precedence than arithmetic operators. Precedence rules save many, many parentheses.

Outline

STS Software Technology Systems

Backus-Naur Form (BNF)

http://news.ku.dk/all_news/2016/famous-danish-computer-scientist-professor-peter-naur-dead/

The Backus-Naur Form is a notation for (context-free) grammars.

- Named for John Backus (IBM) and Peter Naur.
- Notation

::=	"rewrites to"
|	alternative
<>	non-terminals

- In practice, BNF is not strictly obeyed.

STS Software Technology Systems TUHH

Syntax of BNFC files

Recall that BNFC specifies grammars using *labeled* BNF:

Label . Category ::= Production ;

- Productions consist of terminals and non-terminals. Terminals are represented as strings, nonterminals are symbols (identifiers).
  - EAdd. Exp ::= Exp "+" Exp1 ;
- Labels and categories are identifiers.
  - BNFC has no restrictions on identifiers, but the target languages may have.
  - In most programming languages, an identifier is defined as a non-empty sequence of letters, digits, and underscores, starting with a capital letter (see lecture on lexers).

STS Software Technology Systems

TUHH

Example (BNFC grammar)

Here is a grammar for the language of integer arithmetic (assume `Integer` is predefined)

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EDiv. Exp1 ::= Exp1 "/" Exp2 ;
EInt. Exp2 ::= Integer ;
```

Precedences

In BNFC, precedence levels are encoded as digits in the categories themselves:

- Exp1 has precedence level 1.
- Exp2 has precedence level 2.
- Exp is short for Exp0 (level 0) ("base"). It denotes the lowest precedence.

STS Software Technology Systems TUHH

Example (precedences)

- EAdd. Exp ::= Exp "+" Exp1 ;
  - EAdd constructs an expression of level 0 from two expressions of level 0 and level 1.
- EMul. Exp1 ::= Exp1 "*" Exp2 ;
  - EMult constructs an expression of level 1 from two expressions of level 1 and level 2.

Coercions

The highest precedence level is specified using the `coercions` command. It generates BNFC rules that allow translating between different levels.

Example: `coercions Exp 2;` generates the following rules:

```
_. Exp0 ::= Exp1 ;
_. Exp1 ::= Exp2 ;
_. Exp2 ::= "(" Exp0 ")" ;
```

where Exp abbreviates Exp0 and the underscore is a dummy label.

Semantics of precedence

- Instances of categories of different precedence levels all have the same type: the type of the base.
  - 2, 2 + 2, 2 * 2 are all of type Exp
- Expressions of higher precedence levels can be used as subtypes of expressions of lower levels.
  - $2 + 3 = 2_{\mathsf{Exp2}} + 3_{\mathsf{Exp2}}$ type-checks
- Parentheses can be used to lift a lower-level expression to the highest precedence level.
  - $(1 + 2)$ has level 2.

Coercion and correctness

The previous example fails to parse without coercion rules:

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EDiv. Exp1 ::= Exp1 "/" Exp2 ;
EInt. Exp2 ::= Integer ;

-- coercions Exp 2;

> bnfc -m CalcMod.cf
> make
> echo "1 + 2*3" | ./TestCalcMod

Parse              Failed...

Tokens:
[PT (Pn 0 1 1) (TI "1"),PT (Pn 2 1 3) (TS "+" 2),PT (Pn 4 1 5) (↩
    TI "2"),PT (Pn 5 1 6) (TS "*" 1),PT (Pn 6 1 7) (TI "3")]
syntax error at line 1 before 1 + 2 *
```

List categories

Many syntactic categories are used as lists. Example:

- The body of a function is a list of statements.
- The declaration block contains a list of declarations.

BNCF supports lists through the shortcut notation `[]` and two macros, `terminator` and `separator`.

List notation

- Ex.: `[Stm]` denotes a list of `Stm`, `[Def]` denotes a list of `Def`.
- Internally, two rules exist

```
[] .    ListStm  ::=  ;
(:) .   ListStm  ::=  Stm ";" ListStm ;
```

STS Software Technology Systems

Terminators

A *terminator* is a *token* that comes after every list item. For example, a grammar might define that each (function) definition or each statement is terminated by a semicolon.

- The BNFC macro `terminator` can be used with or without the qualifier `nonempty`. No terminator corresponds to the empty string.

```
terminator Stm ``;'' ;
terminator nonempty Stm ``;''
```

- The non-qualified case expands internally to the following two rules:

```
[]   . ListStm ::= ;
(:)  . ListStm ::= Stm ``;'' ListStm ;
```

- The qualifier `nonempty` establishes a singleton list as base.

```
(:[])  . ListStm ::= Stm ``;'' ;
(:)    . ListStm ::= Stm ``;'' ListStm ;
```

STS Software Technology Systems TUHH

Separators

A *separator* is a token that comes in between two items of a list. For example, a grammar might define that two definitions are separated by a semicolon.

BNFC provides a macro `separator`, similar to `terminator`:

```
separator Stm ``;'' ;
separator nonempty Stm ``;''
```

- The non-qualified case expands internally to the following rules:

```
[]     . ListStm ::= ;
(:[])  . ListStm ::= Stm ;
(:)    . ListStm ::= Stm ``;'' ListStm ;
```

- The qualifier `nonempty` expands to two rules:

```
(:[])   . ListStm ::= Stm ;
(:)     . ListStm ::= Stm ``;'' ListStm ;
```

Outline

1. Grammars
   - BNFC
   - A grammar for CPP
   - Abstract syntax

A grammar for CPP

Natural language description

A program is a sequence of definitions. A function definition has a type, a name, an argument list, and a body. An argument list is a comma-separated list of argument declarations enclosed in parentheses (and). A function body is a list of statements enclosed in curly brackets { and }.

An argument declaration has a type and an identifier. Any expression followed by a semicolon ; can be used as a statement. Any declaration followed by a semicolon ; can be used as a statement. Declarations have one of the following formats:

- a type and one variable (as in function parameter lists)
- a type and many variables
- a type and one initialized variable

STS Software Technology Systems

Natural language description (2)

Statements also include
- Statements returning an expression.
- "while" loops, with an expression in parentheses followed by a statement.
- Conditionals: "if" with an expression in parentheses followed by a statement, "else", and another statement.
- Blocks: any list of statements (including the empty list) between curly brackets.

Expressions are specified with the following table that gives their precedence levels. Infix operators are assumed to be left-associative, except assignments, which are right-associative. The arguments in a function call can be expressions of any level. Otherwise, the subexpressions are assumed to be one precedence level above the main expression.

A program may contain comments, which are ignored by the parser. Comments can start with the token // and extend to the end of the line. They can also start with /* and extend to the next */.

Programs

Natural language description

"A program is a sequence of definitions."

BNFC:

PDefs . Program ::= [Def] ;
terminator Def "" ;

Function definitions

"A function definition has a type, a name, an argument list, and a body. An argument list is a comma-separated list of argument declarations enclosed in parentheses (and). A function body is a list of statements enclosed in curly brackets { and } ."

```
int foo (double x, int y) {
    return y + 9;
}
```

BNFC:

DFun . Def ::= Type Id "(" [Arg] ")" "{" [Stm] "}" ;
separator Arg "," ;
terminator Stm "" ;

STS Software Technology Systems

Argument declarations

"An argument declaration has a type and an identifier."

```
double d
int my
```

BNFC:

$$\text{ADecl} \quad . \quad \text{Arg} \quad ::= \quad \text{Type Id} \ ;$$

STS Software Technology Systems

Expressions as statements

"Any expression followed by a semicolon ; can be used as a statement"

```
y + a;
```

BNFC:

$$\text{SExp} \quad . \quad \text{Stm} \quad ::= \quad \text{Exp} \text{ ";" } \quad ;$$

Declarations

"Any declaration followed by a semicolon ; can be used as a statement. Declarations have one of the following formats:

- a type and one variable (as in function parameter lists),
- a type and many variables,
- a type and one initialized variable.

```
int i;
int i, j;
int i = 6;
```

BNFC:

$$
\begin{array}{lcllll}
\text{SDecl} & . & \text{Stm} & ::= & \text{Type Id ";"} & ; \\
\text{SDecls} & . & \text{Stm} & ::= & \text{Type Id "," [Id] ";"} & ; \\
\text{SInit} & . & \text{Stm} & ::= & \text{Type Id "=" Exp ";"} & ;
\end{array}
$$

Statements

"Statements also include

- Statements returning an expression. Ex.: return i + 9 ;
- While loops, with an expression in parentheses followed by a statement. Ex.: while (i < 10) ++i ;
- Conditionals: if with an expression in parentheses followed by a statement, else, and another statement. Ex.: if (x > 0) return x ; else return y ;
- Blocks: any list of statements (including the empty list) between curly brackets."

BNFC:

```
SReturn  .  Stm  ::=  "return" Exp ";" ;
SWhile   .  Stm  ::=  "while" "(" Exp ")" Stm ;
SBlock   .  Stm  ::=  "{" [Stm] "}" ;
SIfElse  .  Stm  ::=  "if" "(" Exp ")" Stm "else" Stm;
```

Expressions

"Expressions are specified with the following table that gives their precedence levels. Infix operators are assumed to be left-associative, except assignments, which are right-associative. The arguments in a function call can be expressions of any level. Otherwise, the subexpressions are assumed to be one precedence level above the main expression."

Expressions (cont'd)

level	expression forms	explanation
15	literal	literal (integer, float, string, boolean)
15	identifier	variable
15	`f(e,...,e)`	function call
14	`v++`, `v--`	post-increment, post-decrement
13	`++v`, `--v`	pre-increment, pre-decrement
13	`-e`	numeric negation
12	`e*e`, `e/e`	multiplication, division
11	`e+e`, `e-e`	addition, subtraction
9	`e<e`, `e>e`, `e>=e`, `e<=e`	comparison
8	`e==e`, `e!=e`	(in)equality
4	`e&&e`	conjunction
3	`e||e`	disjunction
2	`v=e`	assignment

Expressions (cont'd)

```
EMul.   Exp12  ::=  Exp12 "*"  Exp13 ;
EDiv.   Exp12  ::=  Exp12 "/"  Exp13 ;
EAdd.   Exp11  ::=  Exp11 "+"  Exp12 ;
ESub.   Exp11  ::=  Exp11 "-"  Exp12 ;
ELt.    Exp9   ::=  Exp9 "<"  Exp10 ;
EGt.    Exp9   ::=  Exp9 ">"  Exp10 ;
ELEq.   Exp9   ::=  Exp9 "<="  Exp10 ;
EGEq.   Exp9   ::=  Exp9 ">="  Exp10 ;
EEq.    Exp8   ::=  Exp8 "=="  Exp9 ;
ENEq.   Exp8   ::=  Exp8 "!="  Exp9 ;
EAnd.   Exp4   ::=  Exp4 "&&"  Exp5 ;
EOr.    Exp3   ::=  Exp3 "||"  Exp4 ;
EAss.   Exp2   ::=  Exp3 "="  Exp2 ;
```

Expressions (cont'd)

EInt.	Exp15	::=	Integer ;
EDouble.	Exp15	::=	Double ;
EString.	Exp15	::=	String ;
ETrue.	Exp15	::=	"true" ;
EFalse.	Exp15	::=	"false" ;
EId.	Exp15	::=	Id ;
ECall.	Exp15	::=	Id "(" [Exp] ")" ;
EPIncr.	Exp14	::=	Exp15 "++" ;
EPDecr.	Exp14	::=	Exp15 "--" ;
EIncr.	Exp13	::=	"++" Exp14 ;
EDecr.	Exp13	::=	"--" Exp14 ;
ENeg.	Exp13	::=	"-" Exp14 ;

coercions Exp 15 ;

separator Exp "," ;

Comments

"A program may contain comments, which are ignored by the parser. Comments can start with the token // and extend to the end of the line. They can also start with /* and extend to the next */. "

BNFC:

```
comment "//" ;
comment "/*" "*/";
```

- `comment` is a special command. It takes one or two strings.
- The first string indicates the start of a comment; the second string (if available) indicates the end.
- `comment` collaborates with the lexer and instructs it to take its arguments as comments in the source language.

STS Software Technology Systems

TUHH

Types

The category `Type` and the type identifiers need to be specified as well.
BNFC:

Tbool.	Type	::=	"bool" ;
Tdouble.	Type	::=	"double" ;
Tint.	Type	::=	"int" ;
Tstring.	Type	::=	"string" ;
Tvoid.	Type	::=	"void" ;

Identifiers

"An identifier is a letter followed by a list of letters, digits, and underscores."

BNFC:

token Id (letter (letter | digit | '_')*) ;

Identifiers are special.

- They are defined as a *regular expression*.
- BNFC provides the keyword `token` (see lecture on lexical analysis).

STS Software Technology Systems TUHH

Example (work flow)

```
int foo (double x, int y)  {
     return y + 9;
}

> bnfc -m CPP.cf
> make
> ./TestCPP mytest.cpp
mytest.cpp

Parse Successful!

[Abstract Syntax]

PDefs [DFun Type_int (Id "foo") [ADecl Type_double (Id "x"),
ADecl Type_int (Id "y")] [SReturn (EPlus (EId (Id "y"))
(EInt 9))]]

[Linearized tree]

int foo (double x, int y){
     return y + 9 ;
}
```

Grammars of programming languages

Grammars of mainstream languages group productions in categories.
Standard categories include

- Statements
- Expressions
- Declarations
- Keywords

Other categories

- (C++014): Classes, templates, exception handling, ...
- (Haskell): Layout, literate documentation, module, ...

STS Software Technology Systems

Statements

C++0x14: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf

A.5 Statements [gram.stmt]

statement:
labeled-statement
attribute-specifier-seq$_{opt}$ *expression-statement*
attribute-specifier-seq$_{opt}$ *compound-statement*
attribute-specifier-seq$_{opt}$ *selection-statement*
attribute-specifier-seq$_{opt}$ *iteration-statement*
attribute-specifier-seq$_{opt}$ *jump-statement*
declaration-statement
attribute-specifier-seq$_{opt}$ *try-block*

labeled-statement:
attribute-specifier-seq$_{opt}$ *identifier* `:` *statement*
attribute-specifier-seq$_{opt}$`case` *constant-expression* `:` *statement*
attribute-specifier-seq$_{opt}$`default` `:` *statement*

expression-statement:
expression$_{opt}$`;`

compound-statement:
`{` *statement-seq*$_{opt}$`}`

statement-seq:
statement
statement-seq statement

selection-statement:
`if` `(` *condition* `)` *statement*
`if` `(` *condition* `)` *statement* `else` *statement*
`switch` `(` *condition* `)` *statement*

condition:
expression
attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator* `=` *initializer-clause*
attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator braced-init-list*

§ A.5 1208

STS Software Technology Systems

Expressions

http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf, C++0x14

A.4 Expressions **[gram.expr]**

primary-expression:
  literal
  this
  (*expression*)
  id-expression
  lambda-expression

id-expression:
  unqualified-id
  qualified-id

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  literal-operator-id
  ~ *class-name*
  ~ *decltype-specifier*
  template-id

qualified-id:
  nested-name-specifier **template**$_{opt}$ *unqualified-id*

nested-name-specifier:
  ::
  type-name ::
  namespace-name ::
  decltype-specifier ::
  nested-name-specifier identifier ::
  nested-name-specifier **template**$_{opt}$ *simple-template-id* ::

lambda-expression:
  lambda-introducer lambda-declarator$_{opt}$ *compound-statement*

lambda-introducer:
  [*lambda-capture*$_{opt}$]

lambda-capture:
  capture-default
  capture-list
  capture-default , *capture-list*

capture-default:
  &

STS Software Technology Systems TUHH

Declarations

http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf, C++0x14

A.6 Declarations [gram.dcl]

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  explicit-instantiation
  explicit-specialization
  linkage-specification
  namespace-definition
  empty-declaration
  attribute-declaration

block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive
  static_assert-declaration
  alias-declaration
  opaque-enum-declaration

alias-declaration:
  using *identifier attribute-specifier-seq*$_{opt}$ = *type-id* ;

simple-declaration:
  decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$;
  attribute-specifier-seq decl-specifier-seq$_{opt}$ *init-declarator-list* ;

static_assert-declaration:
  static_assert (*constant-expression* , *string-literal*) ;

empty-declaration:
  ;

STS Software Technology Systems

Outline

What is abstract syntax?

We distinguish between abstract syntax and concrete syntax.
Abstract syntax is defined relative to concrete syntax.

- It captures the structure of a sentence (fragment).
  - Category and subcategories
- It unifies the order of subcategories, and ignores the concrete "look."

Example: the following *concrete* expressions all have the same *abstract* syntax

2 + 3	Java, C
(+ 2 3)	Lisp, Scheme
(2 3 +)	Desktop calculator
the sum of 2 and 3	English
icmp 2 3	LLVM

Abstract syntax in BNCF

In BNFC, concrete and abstract syntax are intertwined. A rule itself specifies the concrete syntax, but the abstract syntax can be obtained straightforwardly:

- Obtain the abstract syntax from the concrete syntax by removing terminals, the digits for precedence levels, and coercion rules.
- Example:

Concrete:	EAdd.	Exp0	::=	Exp0	"+"	Exp1
Abstract:	EAdd.	Exp	::=	Exp	Exp	

  The abstract syntax abstracts from the symbol for addition and from precedence levels.

STS Software Technology Systems TUHH

Example (abstract syntax)

Recall the grammar of integer arithmetic (assume `Integer` is predefined)

```
EAdd.  Exp   ::=  Exp "+" Exp1 ;
ESub.  Exp   ::=  Exp "-" Exp1 ;
EMul.  Exp1  ::=  Exp1 "*" Exp2 ;
EDiv.  Exp1  ::=  Exp1 "/" Exp2 ;
EInt.  Exp2  ::=  Integer ;
```

The corresponding abstract BNFC syntax:

```
EAdd.  Exp  ::=  Exp Exp ;
ESub.  Exp  ::=  Exp Exp ;
EMul.  Exp  ::=  Exp Exp ;
EDiv.  Exp  ::=  Exp Exp ;
EInt.  Exp  ::=  Integer ;
```

STS Software Technology Systems TUHH

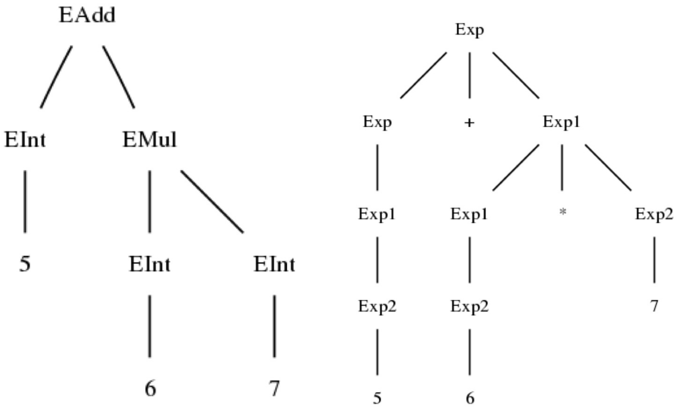
Abstract syntax trees in BNFC

Recall the categories we have seen already: expression, statement, declaration, program.

- A parser generates an abstract syntax tree of the type of Category.
  - The root node is the label of the top production.
  - Inner nodes are labels of the corresponding subtrees.
- Tree labels can be considered *constructors* (of a category).
- A *concrete* parse tree looks different: inner nodes are category symbols, leaves are terminals.

Example (abstract and concrete parse trees)

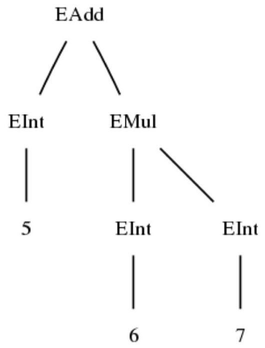
2-slides-ipl-book, slide 30



Example (linearization)

The linearized version of any tree can be obtained by pre-order traversal:

```
EAdd (EInt 5) (EMul (EInt 6) (EInt 7))
```



Representing abstract syntax: Haskell

In Haskell, the abstract syntax of the Calc grammar is implemented as an algebraic data type:

```
data Exp =
      EAdd Exp Exp
    | ESub Exp Exp
    | EMul Exp Exp
    | EDiv Exp Exp
    | EInt Integer
```

- `EAdd, ESub, EMul, EDiv, EInt` are called constructors.
- In general, there is one algebraic data type per category.

STS Software Technology Systems TUHH

Representing abstract syntax: C++ or Java

In C++ or Java, each category and each constructor is represented as a class:

- A category is represented as an abstract base class.
- A constructor is represented as a derived class.

STS Software Technology Systems

Example: abstract syntax in C++

```
class Exp {
public:
  virtual Exp *clone() const = 0;
};
class EAdd : public Exp {
public:
  Exp *exp_1;
  Exp *exp_2;

  EAdd(const EAdd &);
  EAdd &operator=(const EAdd &);
  EAdd(Exp *p1, Exp *p2);
  ~EAdd();
};
class EInt : public Exp {
public:
  Integer integer_;

  EInt(const EInt &);
  EInt &operator=(const EInt &);
  EInt(Integer p1);
  ~EInt();
};
```

Summary

- Parse trees, ambiguities, precedences; BNFC notation
- Abstract syntax, abstract syntax tree (AST)
- Representation of abstract syntax: algebraic data type (FP), class hierarchy (OOP)

STS Software Technology Systems

References

- IPL, Ch. 2
- 2-slides-IPL