Anhang I:
C++ Syntax

Syntaxnotation

Die im folgenden verwendete Notation zur Beschreibung der Syntax einer Programmiersprache nennt sich EBNF–Notation (Erweiterte Backus–Naur–Form). Diese Form der Syntaxbeschreibung ist weit verbreitet und – nach kurzer Einarbeitung – leicht zu lesen und zu verstehen. Im folgenden wird eine Variante von EBNF eingeführt; es gibt viele Varianten. Eine Syntaxbeschreibung (Grammatik) in EBNF besteht aus einer Menge von Regeln. Regeln haben eine linke und eine rechte Seite, die durch die Zeichen ::= getrennt sind. Auf der linken Seite steht ein sogenanntes Nichtterminalsymbol. Ein Nichtterminalsymbol faßt einen bestimmten Teil einer Sprache zusammen, z.B. gibt es in der C++–Grammatik das Nichtterminalsymbol statement, das für eine Anweisung in einem C++–Programm steht. Die Bedeutung ist, daß an all den Stellen, an denen in der Grammatik das Nichtterminal statement verwendet wird, eine Anweisung in einem Programm stehen kann.

Auf der rechten Seite einer Regel steht, wie der Teil der Sprache für das Nichtterminalsymbol auf der linken Regelseite aufgebaut ist. Dazu werden wiederum andere Nichtterminalsymbol und auch Terminalsymbole der Grammatik verwendet. Terminalsymbole sind Zeichen oder Zeichenfolgen, die in einem Programm genau so auftauchen, wie sie in der Grammatik stehen. Um Terminalsymbole von Nichtterminalsymbolen besser unterscheiden zu können, werden sie in einer anderen Schriftart dargestellt. Bei uns ist das der Zeichensatz `Courier`, während Nichtterminalsymbole in Helvetica gesetzt sind.

Auf der rechten Seite einer Regel werden Terminal– und Nichtterminalsymbole miteinander verknüpft. In der regulären EBNF gibt es fünf verschiedene Arten der Verknüpfung:

1. Sequenz: Durch Hintereinanderschreiben von Zeichen und Wörtern gibt man an, daß diese Teile der Sprache auch in einem Programm hintereinander auftauchen.
2. Alternative: Mit dem Zeichen | werden Alternativen gekennzeichnet. Taucht in einer Regeln zum Beispiel a | b auf, dann steht in einem Programm entweder a oder b, aber nicht beides.
3. Option: Durch Klammerung mit eckigen Klammern [und] gibt man an, daß der Teil der Regel in Klammern optional ist. Er kann in einem Programm an der Stelle stehen, muß aber nicht.
4. Wiederholung: Die Klammerung mit geschweiften Klammern { und } gibt an, daß der Teil der Beschreibung in den Klammern an dieser Stelle null bis beliebig oft mal wiederholt werden kann.
5. Nichtleere Wiederholung: Durch Nachstellen eines $^+$ geben wir bei der Wiederholung an, daß der Inhalt der Klammern mindestens einmal im Programmtext auftauchen soll.
6. Gruppierung: Mit runden Klammern (und) werden Gruppen gebildet, wie man das vom Umgang mit Formeln gewohnt ist.
7. Rekursion: Die Rekursion wird nicht besonders gekennzeichnet. Bei der Rekursion werden Nichtterminalsymbole verwendet, deren rechte Regelseiten direkt oder indirekt wiederum das Nichtterminalsymbol verwenden, zu dessen Erklärung sie gerade benutzt werden. Ein Beispiel dafür geben wir unten an.

C++–Grammatik

Die folgende Grammatik gibt nicht den gesamten C++–Sprachumfang wieder. Sie dient aber als Grundlage für die in der Vorlesung verwendete Teilmenge von C++. An vielen Stellen ist die Grammatik restriktiver als es C++ zuläßt. So haben wir insbesondere dem Anweisungsteil eine Struktur aufgeprägt, wie sie auch in anderen Programmiersprachen zu finden ist. Die Art, wie die rechte Regelseite aufgeschrieben ist, gibt in der Regel an, wie wir uns das Layout des entsprechenden Programmteils vorstellen.

Schlüsselwörter

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

Lexikalische Syntax

identifier ::= letter { nondigit | digit }

literal ::= integer_literal | character_literal | floating_literal
| string_literal | boolean_literal

integer_literal ::= dec_int_literal [integer_suffix]
| hex_int_literal [integer_suffix]
| oct_int_literal [integer_suffix]

dec_int_literal ::= nonzero_digit { digit }

hex_int_literal ::= `0x`{hexadecimal_digit}$^{+}$

oct_int_literal ::= `0` { octal_digit }

integer_suffix ::= [l | L][u | U]

letter ::= a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t | u
| v | w | x | y | z | A | B | C | D | E | F
| G | H | I | J | K | L | M | N | O | P | Q
| R | S | T | U | V | W | X | Y | Z

nondigit ::= _ | letter

digit ::= 0 | non_zero_digit

non_zero_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

octal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal_digit ::= digit | a | b | c | d | e | f
| A | B | C | D | E | F

character_literal ::= c_char

floating_literal ::= {digit}$^+$.{digit}$^+$ [e [sign] {digit}$^+$] [floating_suffix]

sign ::= + | –

floating_suffix ::= f | F | l | L

string_literal ::= “{s_char}”

s_char ::= graphic_char außer \, ” und eol | escape_sequence

c_char ::= graphic_char außer \, und eol | escape_sequence

graphic_char ::= Die Menge der darstellbaren Zeichen

escape_sequence ::= simple_esc_seq | oct_esc_seq | hex_esc_seq

simple_esc_seq ::= \” | \? | \\ | \a | \b | \f | \n | \r | \t | \v

oct_esc_seq ::= \ octal_digit octal_digit octal_digit

hex_esc_sequence ::= \x { hexadecimal_digit }$^+$

boolean_literal ::= **true** | **false**

Ausdrücke

expression ::= relation_expression { logical_op relation_expression }

logical_op ::= | | | &&

relation_expression ::= bit_expression [relation bit_expression]

relation ::= == | != | < | > | <= | >=

bit_expression ::= add_expression { bit_op add_expression }

bit_op ::= & | ^ | | | << | >>

add_expression ::= mult_expression { add_op mult_expression }

add_op ::= + | –

mult_expression ::= unary_expression { mult_op unary_expression }

mult_op ::= * | / | %

unary_expression ::= [unary_operator] simple_expression
| new_expression | delete_expression

unary_operator ::= * | & | + | – | ! | ~

new_expression ::= **new** identifier { [expression] } [(expression_list)]

expression_list ::= expression {, expression}

delete_expression ::= **delete** [[]] simple_expression

simple_expression ::= primary_expression | cast_expression | typeid_expression

primary_expression ::= literal | **this** | (expression) | qualified_name

qualified_name ::= {*class_or_namespace_*identifier ::} name

name ::= identifier | operator_symbol | indexed_component
| selected_component | function_call

operator_symbol ::= **operator** operator

operator ::= logical_op | bit_op | relation | add_op | mult_op

indexed_component ::= name [expression]

selected_component ::= name . identifier | name –> identifier

function_call ::= name([expression_list])

cast_expression ::= **dynamic_cast** < *type_*identifier > (expression)
| **static_cast** < *type_*identifier > (expression)
| **reinterpret_cast** <*type_*identifier> (expression)
| **const_cast** < *type_*identifier > (expression)

typeid_expression ::= **typeid** (expression)

Anweisungen

statement_list ::= { statement; }+

statement ::= [identifier :] simple_statement
| [identifier :] compound_statement

simple_statement ::= assignment_statement | jump_statement
| throw_statement | inc_statement
| dec_statement | call_statement

compound_statement ::= block | if_statement | switch_statement | while_statement
| do_while_statement | for_statement | try_block

assignment_statement ::= identifier assignment_operator expression

assignment_operator ::= = | *= | /= | %= | += | -= | >>= | <<= | &=
| ^= | |=

block ::= {
declaration_list
statement_list
}

if_statement ::= **if** (expression) {
statement_list}
{**else if** (expression) {
statement_list } }
[**else** {
statement_list
}]

switch_statement ::= **switch** (expression) {
{case_list statement_list **break**;}
[**default**: statement_list]
}

case_list ::= { **case** *const*_expression : }+

while_statement ::= **while** (expression) {
statement_list
}

do_while_statement ::= **do** {
statement_list }
while (expression)

for_statement ::= **for** (*discrete*_init_decl ; condition ; inc_dec) {
statement_list
}

init_decl ::= *type*_identifier init_object

inc_dec ::= inc_statement | dec_statement

jump_statement ::= **break** | **continue** | **return** [expression]
| **goto** *label*_identifier

try_block ::= **try** block handler_sequence

handler_sequence ::= {**catch** (exception_declaration) {
statement_list
} }+

throw_statement ::= **`throw`** [expression]

inc_statement ::= designator++

dec_statement ::= designator−−

call_statement ::= name (expression_list)

Deklarationen

declaration_list ::= { declaration **`;`** }

declaration ::= object_declaration | type_declaratoin
| subprogram_decl | subprogram_definition
| linkage_specification

object_declaration ::= [storage_class] *type*_identifier [cv_spec] init_object_list

storage_class ::= **`auto`** | **`register`** | **`static`** | **`extern`** | **`mutable`**

cv_spec ::= { **`const`** | **`volatile`** }+

init_object_list ::= init_object { **`,`** init_object }

init_object ::= [* | &] identifier [[*constant*_expression]] [initializer]

initializer ::= **`=`** init_expression | (expression_list)

init_expression ::= expression | { [init_expression_list] }

init_expression_list ::= init_expression { **`,`** init_expression }

type_declaration ::= class_declaration | struct_declaration | union_declaration
| enum_declaration | typedef_declaration

class_declaration ::= **`class`** identifier [inheritance_spec]
{
[**`public`**: decration_list]
[**`protected`**: declaration_list]
[**`private`**: declaration_list]
}

inheritance_spec ::= **`:`** inheritance { **`,`** inheritance }

inheritance ::= [**`virtual`**] [access_specification] *class*_identifier

access_specification ::= **`public`** | **`protected`** | **`private`**

struct_declaration ::= **`struct`** identifier {
{ object_declaration **`;`** }
}

union_declaration ::= **`union`** identifier {
{ object_declaration **`;`** }
}

enum_declaration ::= **`enum`** identifier {
enumerator_definition { **`,`** enumerator_definition}
}

enumerator_definition ::= identifier [= *constant*_expression]

typedef_definition ::= **`typedef`** *type*_identifier [* | &] identifier
[[*constant*_expression]]

subprogram_decl ::= [subp_spec] (function_decl | procedure_decl)

subp_spec ::= **`inline`** | **`virtual`** | **`explicit`**

function_decl ::= *type*_identifier qualified_name parameter_spec

procedure_decl ::= **`void`** qualified_name parameter_spec

parameter_spec ::= ([formal_parameter { , formal_parameter }])

formal_parameter ::= *type*_identifier [* | &] identifier [= *constant*_expression]

subprogram_definition ::= subprogram_decl [**`try`**] [ctor_initializer]
block
[handler_sequence]

ctor_initializer ::= **`:`** member_initializer { **`,`** member_initializer }

member_initializer ::= identifier ([expression_list])

linkage_specification ::= **`extern`** string_literal declaration
| **`extern`** string_literal { declaration_list }

Programmstruktur

file ::= program_file | header_file

context_clause ::= **`using namespace`** *namespace*_name **`;`**
| `#include "`*file*_identifier`"`
| `#include <`*file*_identifier`>`

header_file ::= {context_clause} { header_declaration }$^{+}$

program_file ::= {context_clause} {declaration}$^{+}$

header_declaration ::= subprogram_decl | *const*_object_declaration
| type_declaration