Übersetzerbau 2

- Syntaxanalyse
- Codeerzeugung
- Web Assembly Instructions

Syntaxanalyse
Kontextfreie Grammatiken und Kellerautomaten

Kontextfreie Grammatiken

Problem

Reguläre Grammatiken können keine Zentralrekursion ausdrücken

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

Dafür braucht man zumindest kontextfreie Grammatiken

Definition

Eine Grammatik heisst ***kontextfrei*** (KFG), wenn alle Produktionen folgende Form haben:

```
A = a.
```

A ∈ NTS, a beliebige nichtleere Folge von TS und NTS

In EBNF kann a auch |, (), [] und {} enthalten

Beispiel

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

indirekte Zentralrekursion

Kontextfreie Sprachen werden durch ***Kellerautomaten*** erkannt

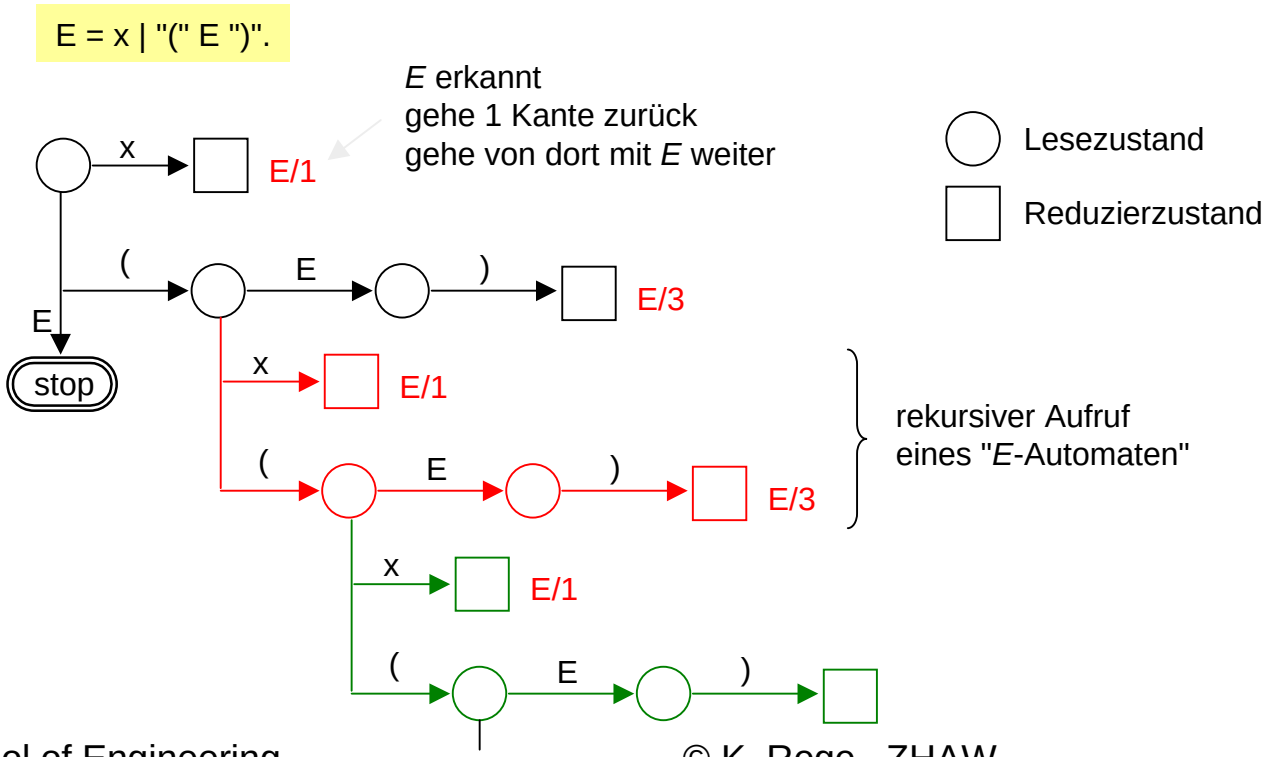
Kellerautomat

engl.: Push Down Automaton (**PDA**)

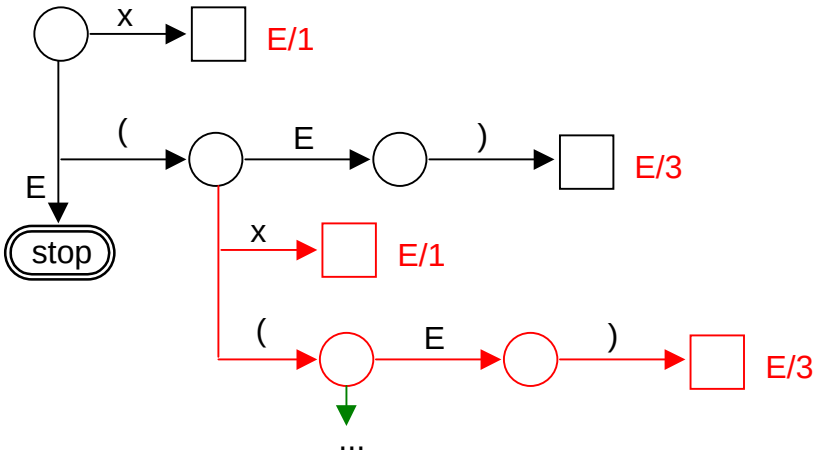
Eigenschaften

- erlaubt Zustandsübergänge mit Terminal- und Nonterminalsymbolen
- merkt sich Zustandsübergänge in einem Keller (Stack)

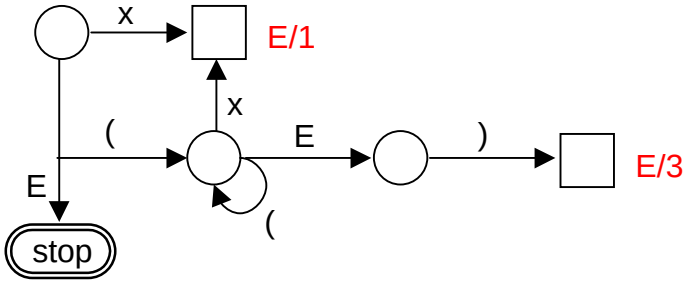
Beispiel



Kellerautomat (Fortsetzung)



Kann vereinfacht werden zu



Erfordert Keller, um den Rückweg durch alle bisher durchlaufenen Zustände zu finden

Einschränkung kontextfreier Grammatiken

KFGs können keine *Kontextbedingungen* ausdrücken

Zum Beispiel:

- ***Jeder verwendete Name muss deklariert worden sein***
  Deklaration gehört zum Kontext der Verwendung
  x = 3;
  kann je nach Kontext richtig oder falsch sein
- ***In Ausdrücken müssen die Typen der Operanden übereinstimmen***
  Typen werden in der Deklaration festgelegt → gehört zum Kontext der Verwendung

Lösungsmöglichkeiten

- ***Kontextsensitive Grammatiken***
  zu kompliziert

- ***Überprüfung der Kontextbedingungen in der semantischen Analyse***
  d.h. die Grammatik erlaubt Konstrukte, die laut Kontextbedingungen verboten sind,
  z.B.:int x = "drei";syntaktisch korrekt
  semantisch falsch
  Fehler wird erst bei semantischer Analyse gemeldet.

Kontextbedingungen

Für jede Syntaxregel werden die semantischen Zusatzbedingungen angegeben

Zum Beispiel

Statement = Designator "=" Expr ";".

- *Designator* muss eine Variable, ein Arrayelement oder ein Objektfeld bezeichnen.
- Der Typ von *Expr* muss mit dem Typ von *Designator* zuweisungskompatibel sein.

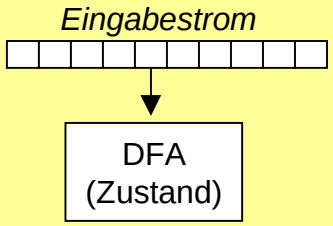
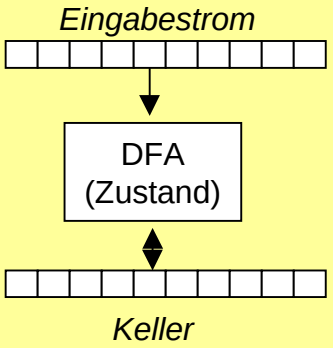
Factor = "new" ident "[" Expr "]".

- *ident* muss einen Typ bezeichnen.
- Der Typ von *Expr* muss *int* sein.

$Designator_1$ = $Designator_2$ "[" Expr "]".

- Der Typ von $Designator_2$ muss ein Arraytyp sein.
- Der Typ von *Expr* muss *int* sein.

Reguläre versus kontextfreie Grammatiken

	Reguläre Grammatiken	Kontextfreie Grammatiken
Anwendung	Lexikalische Analyse	Syntaxanalyse
Erkennung	**DFA** (kein Keller) 	**PDA** (Keller) 
Produktionen	A = a | b C.	A = a.
Probleme	Klammerkonstrukte	Kontextsensitive Konstrukte (z.B. Typprüfungen, ...)

Syntaxanalyse - Rekursiver Abstieg

Syntaxanalyse mit Rekursivem Abstieg

engl.: *Recursive Descent Parsing*

- Topdown-Technik
- Syntaxbaum wird von oben nach unten aufgebaut

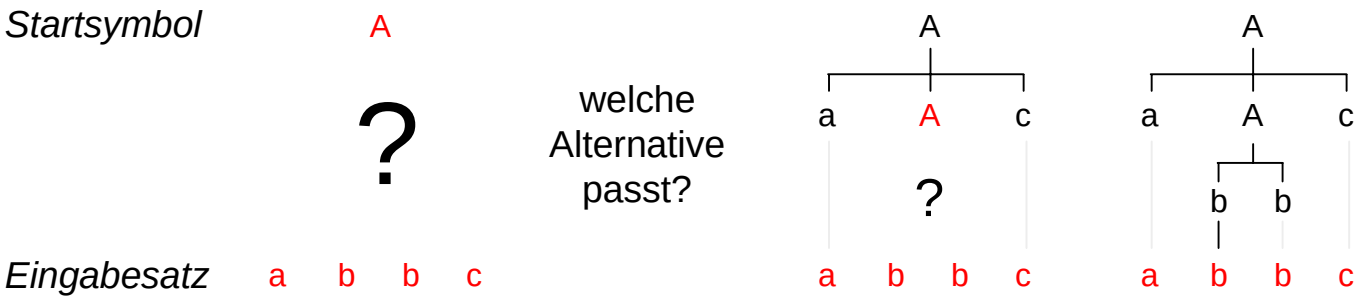
Beispiel

Grammatik

A = a A c | b b.

Eingabesatz

a b b c



Auswahl der passenden Alternative auf Grund von

- Vorgriffssymbol aus dem Eingabestrom
- Terminale Anfänge der einzelnen Alternativen

Statische Variablen des Parsers

Vorgriffssymbol

Parser schaut immer um 1 Symbol voraus (look ahead)

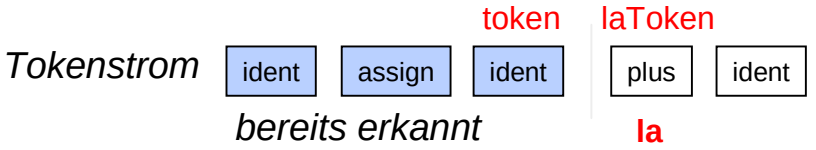
```
static int la;          // Tokencode des Vorgriffssymbols (lookahead)
```

Parser merkt sich die beiden letzten erkannten Symbole (für Semantikverarbeitung)

```
static Token token;     // zuletzt erkanntes Token
static Token laToken;   // lookahead token (noch nicht erkannt)
```

Diese Variablen werden von der Hilfsmethode *Scan()* gefüllt

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



Scan() wird zu Beginn der Syntaxanalyse aufgerufen → erstes Symbol steht in *la*

Erkennung von Terminalsymbolen

Muster

Eingabesymbol: a
Parser-Aktion: Check(a);

Dazu nötige Hilfsmethoden

```
static void check (int expected) throws Exception  {
    if (la == expected) scan();  // erkannt, daher weiterlesen
    else error(  Token.names[expected] + " expected" );
}
```

```
public static void error (string msg) throws Exception {
    System.out.printf("-- line %s, col %s: %s", laToken.line, laToken.col, msg);
    throw new Exception("Panic Mode"); // bessere Lösung siehe später
}
```

in Klasse *Token*:

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

geordnet nach Tokencodes

Namen der Terminalsymbole sind als Konstanten der Klasse *Token* deklariert

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

Erkennung von Nonterminalsymbolen

Muster

Nonterminalsymbol: A
Parser-Aktion: A(); // Aufruf der Erkennungsmethode von A

Jedes Nonterminalsymbol wird durch eine gleichnamige Methode erkannt

```
static void A () {
    ... Aktionen zur Erkennung von A ...
}
```

Initialisierung des Parsers

```
public static void Parse () {
    Scan();                 // füllt token, laToken und la
    Program();              // ruft die Erkennungsmethode des Startsymbols auf
    Check(Token.EOF);       // nach dem Programm muss die Eingabedatei zu Ende sein
}
```

Erkennung von Sequenzen

Muster

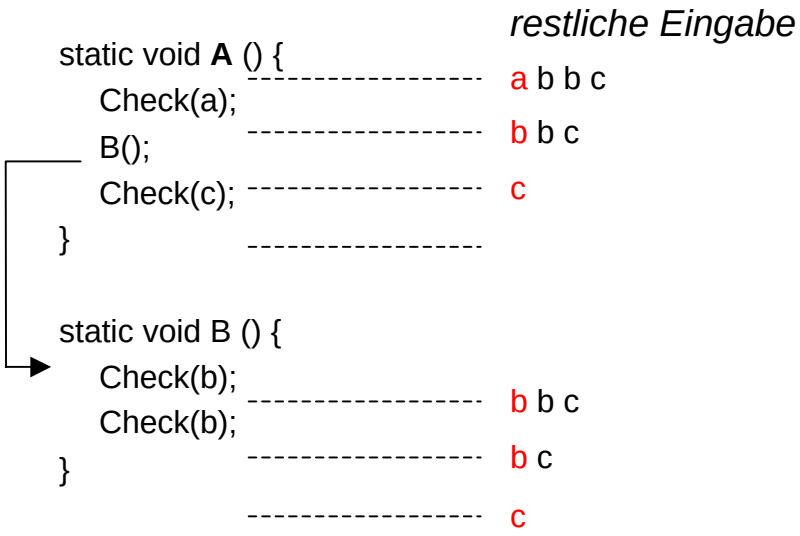
Grammatikregel: A = a B c.

Erkennungsmethode:

```
static void A () {
    // la enthält das erste Symbol von A
    Check(a);
    B();
    Check(c);
    // la enthält das Nachfolgesymbol von A
}
```

Simulation

A = a B c.
B = b b.



Erkennung von Alternativen

Grammatikmuster: a | b | g a, b, g sind beliebige EBNF-Ausdrücke

Erkennungsaktion:

```
if (la ∈ First(a)) { ... erkenne a ... }
else if (la ∈ First(b)) { ... erkenne b ... }
else if (la ∈ First(g)) { ... erkenne g ... }
else Error("...");  // sprechende Fehlermeldung finden
```

Beispiel

```
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");
}
```

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

Erkennung der Phrasen a d und c b
Fehler bei b b

Erkennung von EBNF-Optionen

Grammatikmuster: [a] a ist ein beliebiger EBNF-Ausdruck

Erkennungsaktion: if (la ∈ First(a)) { ... *erkenne* a ... } // kein Fehlerzweig!

Beispiel

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

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

Erkennung der Phrasen a b c und c

Erkennung von EBNF-Iterationen

Grammatikmuster: { a } a ist ein beliebiger EBNF-Ausdruck

Erkennungsaktion: while (la ∈ First(a)) { ... *erkenne* a ... }

Beispiel

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

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

Erkennung der Phrasen a c d c b und a b

Alternativ dazu

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

Ohne EOF: Gefahr einer Endlosschleife, wenn b vergessen wird.

Rekursiver Abstieg und Syntaxbaum

Syntaxbaum wird u.U. nur implizit aufgebaut

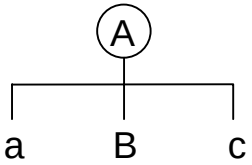
- durch Methoden, die gerade aktiv sind
- d.h. durch die Produktionen, an denen gerade gearbeitet wird

Beispiel

A = a B c.
B = d e.

Aufruf von *A()*

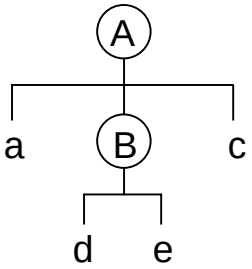
```
static void A () {
    Check(a); B(); Check(c);
}
```



A in Arbeit

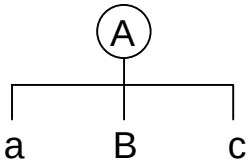
Erkennung von *a*
Aufruf von *B()*

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



A in Arbeit
B in Arbeit

Erkennung von *d* und *e*
Rückkehr von *B()*



A in Arbeit

"Keller"

Beispiel arithmetische Ausdrücke

Grammatik

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

- Beispiele mathematischer Ausdrücke
  - 3 + 3 + 3 oder 3 + 3*2 + 2 oder 3 * (2 + 3)

```
void expr() {
    term();
    while (Scanner.la == Token.PLUS
          || Scanner.la == Token.MINUS) {
      Scanner.scan();
      term();
    }
}
```

```
void term() {
    factor();
    while (Scanner.la == Token.TIMES
          || Scanner.la == Token.SLASH) {
       Scanner.scan();
       factor();
    }
}
```

```
void factor() {
     if (Scanner.la == Token.LBRACK) {
       Scanner.scan();
       expr();
       Scanner.check(Token.RBRACK);
     }
     else if (Scanner.la == Token.NUMBER) {
       Scanner.scan();
     }
     else Scanner.error("illegal Symbol");
 }
```

```
public static void main(String[] args)  {
     Scanner.init("45*(3+4)/78+45-56");
     Scanner.scan();
     expr();
     Scanner.check(Token.EOF);
 }
```

Logikanbindung - Interpreter

Logik kann direkt in Parser hineinkodiert werden

```
static int stack[] = new int[10];
static int sp = 0;

static void push(int val) {
  stack[sp++]=val;
}

static int pop() {
  return stack[--sp];
}
```

+ einfach, effizient

+ Stackrechner sehr einfach

- Vermischung von Syntax und Semantik

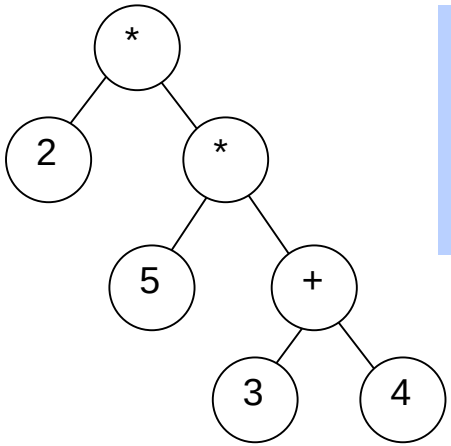
bei Compiler: statt auszuwerten wird Code erzeugt

```
void factor() throws Exception {
   if (Scanner.la == Token.LBRACK) {
      Scanner.scan();
      expr();
      Scanner.check(Token.RBRACK);
   }
   else if (Scanner.la == Token.NUMBER) {
      Scanner.scan();
      push(Scanner.token.val);
    }
   else Scanner.error("illegal Symbol");
 }
```

```
static void expr()  throws Exception {
    term();
    while (Scanner.la == Token.PLUS
      || Scanner.la == Token.MINUS) {
      Scanner.scan();
      int op = Scanner.token.kind;
      term();
      if (op == Token.PLUS)
           push(pop()+pop());
      else push(-pop()+pop());
    }
}
```

Logikanbindung - der abstrakte Syntaxbaum

- Während Parsing wird Datenstruktur erstellt
- Bsp :
  - aus dem Ausdruck 2 * 5* (3 + 4) wird folgender Baum erstellt
- innerer Knoten: Operatoren
- Blätter: Operanden




```
public class Item {
  int val;
  int kind;
  Token token;
  Item left, right;
}
```

```
Item factor() {
    Item item = null;
    if (Scanner.la == Token.LBRACK) {
      Scanner.scan();
      item = expr();
      Scanner.check(Token.RBRACK);
    }
    else if (Scanner.la == Token.NUMBER) {
      Scanner.scan();
      item = new Item(Scanner.token);
    }
    else Scanner.error("illegal Symbol");
    return item;
}
```

```
 Item term() {
    Item item = factor();
    while (Scanner.la == Token.TIMES
        || Scanner.la == Token.SLASH) {
      Scanner.scan();
      Item i = new Item(Scanner.token);
      i.left = item; i.right = factor(); item = i;
    }
    return item;
 }
```

Interpretation des Syntaxbaums

- Der Syntaxbaum kann durch eine rekursive eval-Methode ausgewertet werden
  - Klammerung wurde schon beim Aufbau des Baumes berücksichtigt

```
int eval(Item item) {
  int val = 0;
  if (item.kind == Token.NUMBER) val = item.val;
  else if (item.kind == Token.PLUS) val = eval(item.left)+eval(item.right);
  else if (item.kind == Token.MINUS) val = eval(item.left)-eval(item.right);
  else if (item.kind == Token.TIMES) val = eval(item.left)*eval(item.right);
  else if (item.kind == Token.SLASH) val = eval(item.left)/eval(item.right);
  return val;
}
```

- Vorteile:
  - Trennung zwischen Syntaxanalyse und Berechnung/Codeerzeugung
  - Ausdruck muss nur einmal übersetzt werden und kann dann mehrmals ausgewertet werden
  - Es kann im Ausdruck zuvor nach mehrfach vorkommenden Teilen gesucht werden -> Optimierungen

Syntaxanalyse - Fehlerbehandlung

Ziele der Syntaxfehlerbehandlung

School of Engineering

Anforderungen

1. Der Parser soll möglichst viele Fehler pro Übersetzung finden
2. Der Parser darf auch bei schlimmen Fehlern nicht abstürzen
3. Die Fehlerbehandlung soll die fehlerfreie Analyse nicht bremsen
4. Die Fehlerbehandlung soll den Parser-Code nicht aufblähen

Fehlerbehandlungsmethoden für den rekursiven Abstieg

- Fehlerbehandlung im "Panic Mode"
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

Panic Mode

Die Syntaxanalyse wird nach dem ersten Fehler abgebrochen

```
public static void error(String msg) throws Exception {
        throw new Exception(msg+" at " + Scanner.laToken.pos);
}
```

Vorteile

- billig
- für kleine Kommandosprachen oder für Interpreter ausreichend

Nachteile

- für grössere Sprachen indiskutabel

Fehlerbehandlung mit allgemeinen Fangsymbolen

School of Engineering

Beispiel

erwartete Symbolfolge: a b c d ...
gelesene Symbolfolge: a x y z d ...

Wiederaufsatz (Synchronisation der restlichen Eingabe mit der Grammatik)

1. **"Fangsymbole" bestimmen, mit denen nach dem Fehler fortgesetzt werden kann.**

   Mit welchen Symbolen kann hier fortgesetzt werden?
   mit *c* Nachfolger von *b* (das an der Fehlerstelle erwartet wurde)
   mit *d* Nachfolger von *b c*
   ...
   Fangsymbole sind an dieser Fehlerstelle {*c*, *d*, ...}

2. **Fehlerhafte Symbole überlesen, bis ein Fangsymbol auftritt.**
   x ,*y, z* werden hier überlesen; mit *d* kann fortgesetzt werden.

3. **Parser an die Stelle in der Grammatik steuern, an der fortgesetzt werden kann.**

Beliebige Fangsysmbole realitv komplex in der Implementation; Verlangsamung des Parsers

Fehlerbehandlung mit speziellen Fangsymbolen

Synchronisation erfolgt nur an besonders "sicheren" Stellen

d.h. an Stellen, die mit Schlüsselwörtern beginnen, die an keiner anderen Stelle in der Grammatik vorkommen

Zum Beispiel

Fangsymbolmengen

- Statement-Anfang: if, while, do, ...
- Deklarationsanfang: public, static, void, ...

Code, der an der Synchronisationsstelle eingefügt werden muss

Fangsymbolmenge an dieser Synchronisationsstelle

```
...
if (la ∈ expectedSymbols) {
    Error("...");  // keine Nachfolgermengen; kein Überlesen in Error()
    while (la ∈ (expectedSymbols && != {eof})) Scan();
}
...
```

damit es zu keiner Endlosschleife kommt

- Einfach
- Nach einem Fehler "rumpelt" der Parser bis zur nächsten Synchronisationsstelle weiter

Codeerzeugung - Überblick

Aufgaben der Codeerzeugung

Erzeugung von Maschinenbefehlen

- Auswahl passender Instruktionen
- Auswahl passender Adressierungsarten

Umsetzung von Verzweigungen und Schleifen in Sprünge

Eventuell Optimierungen

Ausgabe der Objektdatei

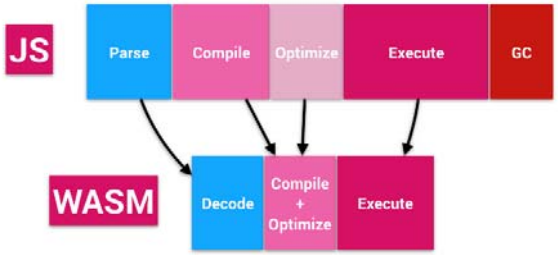
Wie geht man vor?

- **Studieren der Zielmaschine**
  Register, Datenformate, Adressierungsarten, Instruktionen, Befehlsformate, ...
- **Festlegen von Laufzeitdatenstrukturen**
  Layout von Aktivierungssätzen, globalen Daten, Heapobjekten, Stringkonstantenspeicher, ...
- **Verwaltung des Codespeichers**
  Bit-Codierung der Befehle, Patchen des Codes, ...
- **Implementierung der Codeerzeugungsmethoden** (in folgender Reihenfolge)
  - Laden von Werten und Adressen (in Register oder auf den Stack)
  - Verarbeitung zusammengesetzter Bezeichner (x.y, a[i], ...)
  - Übersetzung von Ausdrücken
  - Verwaltung von Sprüngen und Marken
  - Übersetzung von Anweisungen
  - Übersetzung von Methoden und Parametern
- **Registerverwaltung**
  entfällt bei einer VM da diese (meistens) Stackmaschinen sind

Codeerzeugung - WebAssembly Format

What is Web Assembly

- WASM is a **Compiler Target** (code generated by compilers)
- C, C++, Rust, Go (and other) in browsers with ***close to native*** *performance*.
- Compiling JavaScript to native is inherently complex and cumbersome because JavaScript is highly dynamic (e.g. no static types).



- Failed approaches

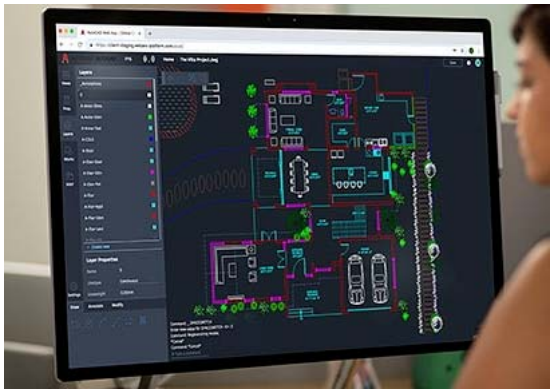
- WASM supported by

… What is Web Assembly

- Applications can be compiled to WASM and run in browser

- Google Earth in Browser

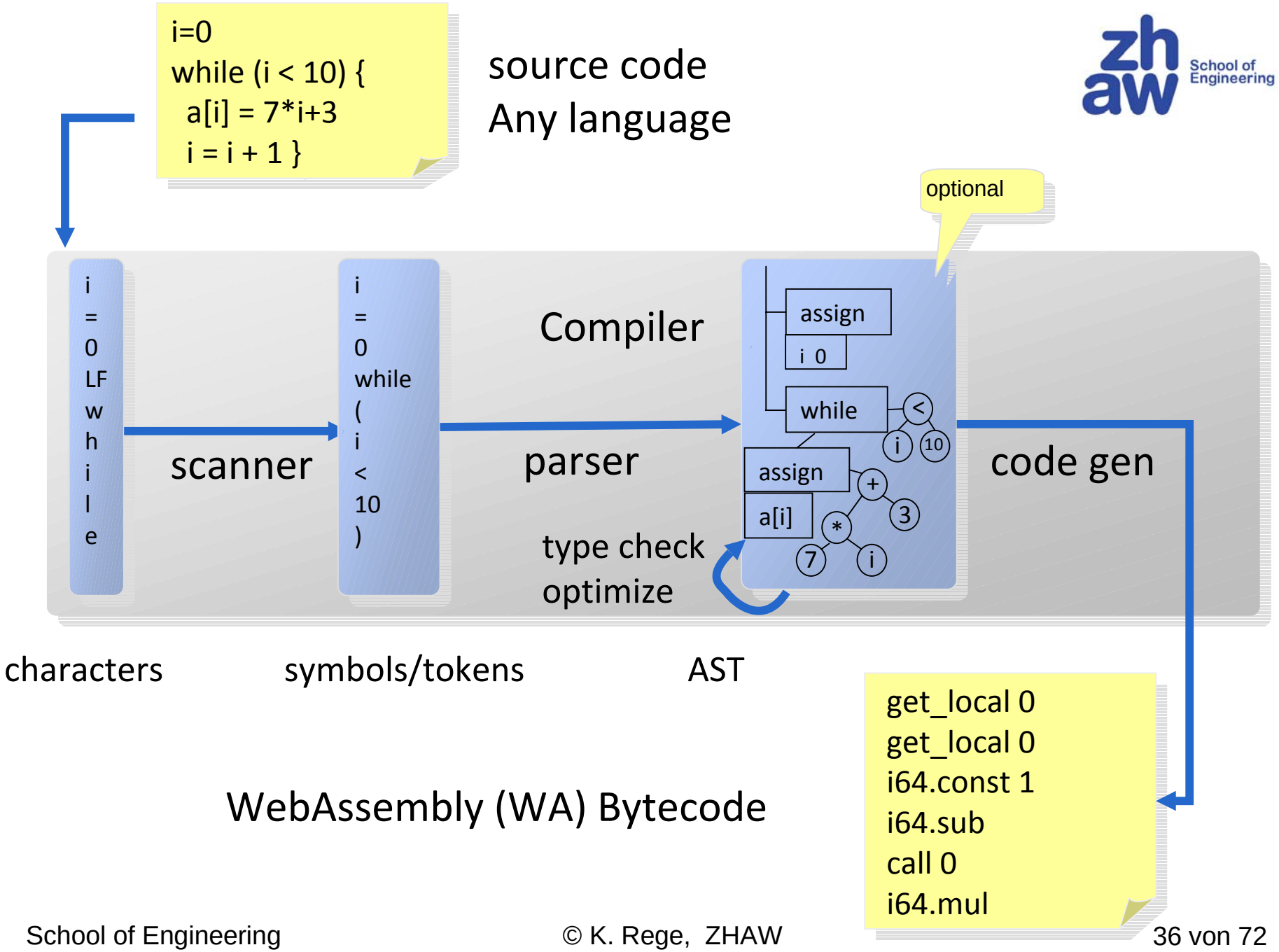
AutoCAD in Browser



https://webassembly.eu/

- Much like Applets[RIP]: NASA World Wind in Browser

https://en.wikipedia.org/wiki/Java_applet


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
Any language
optional
i = 0 LF w h i l e
scanner
i = 0 while (i < 10)
Compiler
parser
type check
optimize
assign
i 0
while
<
i
10
assign
+
a[i]
*
3
7
i
code gen
characters
symbols/tokens
AST
get_local 0
get_local 0
i64.const 1
i64.sub
call 0
i64.mul
WebAssembly (WA) Bytecode

Simple Example

- C/C++ can be compiled to wasm
- Simple example of a recursive factorial function

C input source	Linear assembly bytecode (intermediate representation)	Wasm binary encoding (hexadecimal bytes)

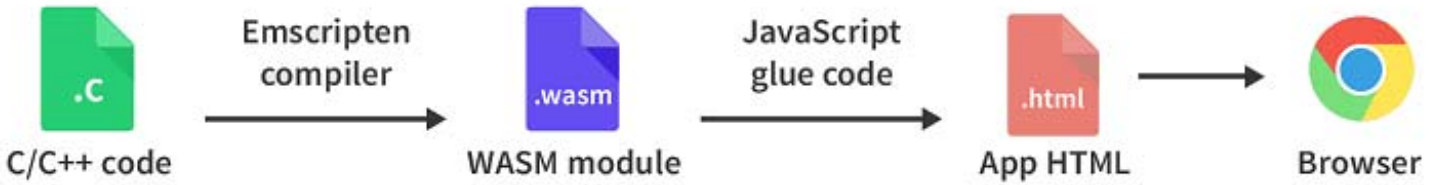
`int factorial(int n) {`
`  if (n == 0)`
`  return 1;`
`  else`
`  return n * factorial(n-1);`
`}`
	`get_local 0`
	`i64.eqz`
	`if (result i64)`
	`  i64.const 1`
	`else`
	`  get_local 0`
	`  get_local 0`
	`  i64.const 1`
	`  i64.sub`
	`  call 0`
	`  i64.mul`
	`end`
		`20 00`
		`50`
		`04 7E`
		`42 01`
		`05`
		`20 00`
		`20 00`
		`42 01`
		`7D`
		`10 00`
		`7E`
		`0B`

… Simple Example

- C/C++ code can be compiled by emcc to a .wasm file
- Compiler generates code for this wa-virtual machine

```
emcc factorial.c -s WASM=1 -O3 -o index.js
```

- + generated js-glue code for browser



… Simple Example: HTML

- Simple Java Script to call WASM

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head></head>
<body>
  <script type="text/javascript">
        fetch("Factorial.wasm")
          .then(response => response.arrayBuffer())
          .then(bytes => WebAssembly.instantiate(bytes, {imports: {}}))
          .then(results => {
            document.getElementById("demo").innerHTML
              = results.instance.exports.factorial(10);
        });
  </script>
  <h1>Test WASM<h1>
  <p id="demo"></p>
</body>
</html>
```

… Simple Example: node.js Sample

- Define Program e.g. in WAST Format (Syntax Tree)

```
(module
  (func (export "addTwo") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

- Call the Program via Node.js

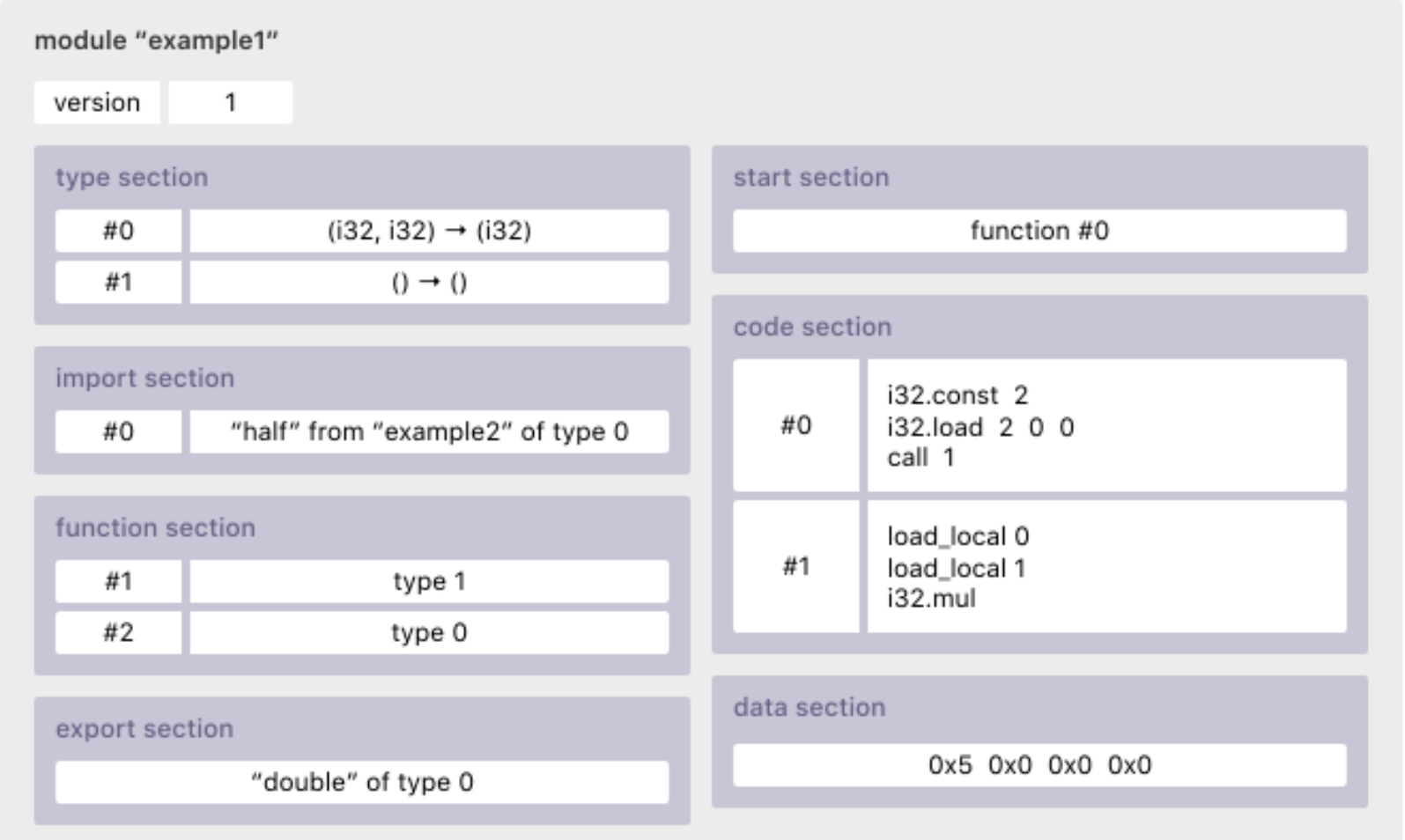
```
const fs = require('fs');
const buf = fs.readFileSync('./addTwo.wasm');
const lib = await WebAssembly.instantiate(new Uint8Array(buf)).
  then(res => res.instance.exports);

console.log(lib.addTwo(2, 2)); // Prints '4'
console.log(lib.addTwo.toString()); // Prints 'function addTwo() { [native code] }'
```

http://thecodebarbarian.com/getting-started-with-webassembly-in-node.js.html

Structure of a WebAssembly File

- A Web Assembly file is structured as follows:



https://rsms.me/wasm-intro

Function Type Section

- list of unique function signature
  - functions declared in module
  - functions imported

- position in the list is the type index within the module

```
(i32 i32 -> i32)        // func_type #0
(i64 -> i64)            // func_type #1
( -> )                  // func_type #2
```

Import Section

- Declares any external dependencies by listing module name
- Field name and type for each function, value or data required

```
("dumb-math", "quadruple", (func_type 1))          // func $0
("dumb-math", "pi", (global_type i64 immutable))
```

Export Section

- Declares any parts of the module that can be accessed by the host environment

Start Section

- Function index for a function to be called when the module is loading

Code - Section

Language Types

- Integer Value Types
  - They may be interpreted as signed or unsigned by individual operations. When interpreted as signed, a two's complement interpretation is used.

Name	Bits	Description
`i32`	32	32-bit integer
`i64`	64	64-bit integer

- Floating-Point Value Types

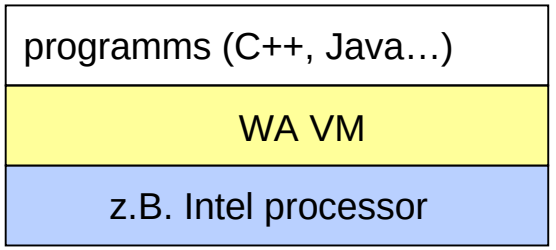
Name	Bits	Description
`f32`	32	IEEE 754-2008 binary32, commonly known as "single precision"
`f64`	64	IEEE 754-2008 binary64, commonly known as "double precision"

- Booleans
  - Are represented as values of type i32.
  - In a boolean context, such as a br_if condition, any non-zero value is interpreted as true and 0 is interpreted as false.

Virtual Machine and Expression Stack

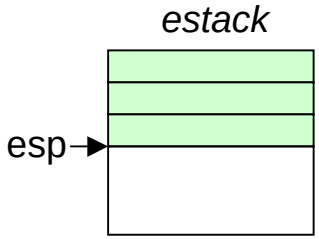
What is a virtual machine (VM)?

- A CPU implemented in software
- Instructions are interpreted or JIT-compiled
- other examples: JVM, CLR, Smalltalk-VM, Pascal P-Code



VMs are often Stack machines

- no registers
- but an ***Expression Stack*** (operands and results on the stack)



esp expression stack pointer

JIT Just in time compilation

- every methode/classe/funktion is compiled to real machine code of that very platform

Execution of a Stack Machine

Beispiel

Expression **i = i + j * 5;**

Values of i and j

Abarbeitung

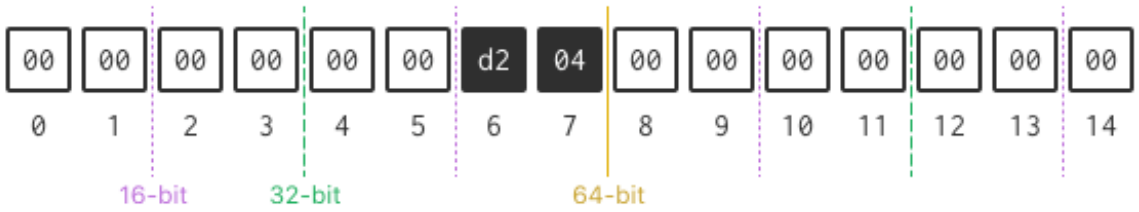
command	*expression stack*	
i32.get_local 0	3	load local variable i (slot 0) onto stack
i32.get_local 1	3 4	load local variable j (slot 1) onto stack
i32.const 5	3 4 5	load const 5 onto stack
i32.mul	3 20	multiplicate 2 topmost stackelemts; result top of stack
i32.add	23	add 2 topmost stackelemts; result top of stack
i32.set_local 0		store topmost stackelemt in i (slot 0)

at the end of the stack is empty

Accessing global Memory

- Access needs two immediate values to the load and store operators
  - Effective-address = address-operand (on stack) + offset-immediate
    - *Alignment immediate is one of the following values: 0 = 8-bit, 1 = 16-bit, 2 = 32-bit, and 3 = 64-bit.*

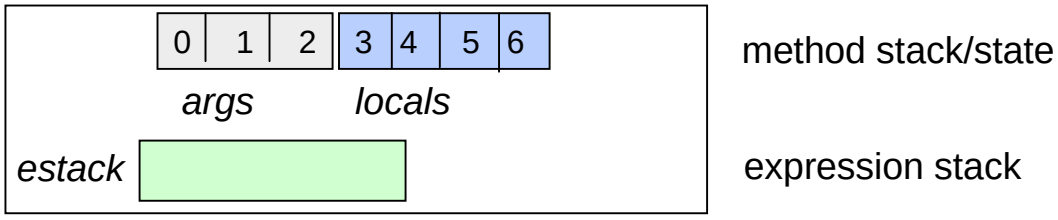
```
i32.const 3       // address-operand = 3
i64.const 1234    // value
i64.store16 1 3   // alignment = 1 = 16-bit, offset-immediate = 3
                  // effective-address = 3 + 3 = 6
```



Accessing local Memory (i.e. Variables, Args.)

Method Stack/State

- manages areas for
  - arguments and local variables (*args und locals*)
  - expression stack (*estack*)
- every method invocation has unique method state
- Method states are organized as stacks
  method area on the stack is also called *Stack Frame*.
- each method parameter and local variables use a slot dependent of variable size
- Adresses are sequential (slot-)numbers, according order of declaratoin
- z.B. i32.get_local 0 loads first method argument stack onto *estack*
  i32.get_local 3 loads value of first local variable onto estack *(3 arguments)*



Instruction set of the WASM

Byte Code (only subset described here)

- very compact: most instructions are only 1 byte long
- instructions for different types

Integer

```
i32.const 0
i32 get_local 1
i32 set_local 2
```

Float

```
f32.const 0
f32 get_local 1
f32 set_local 2
```

Double

```
f64.const 0
f64 get_local 1
f64 set_local 2
```

Instruction format

Simple compared to real cpu

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

opcode ... 1 or 2 byte
operand ... primitive datentype

Example

0 Operanden	`i32.add`	has three implizit operands
1 Operand	`i32.get_local 1`	has one explizit operand

Constant

- The const instruction pushes $value onto the expression stack (typed)

Mnemonic	Opcode	Immediates	Signature	Families
`i32.const`	0x41	`$value` : varsint32	`() : (i32)`	
`i64.const`	0x42	`$value` : varsint64	`() : (i64)`	
`f32.const`	0x43	`$value` : float32	`() : (f32)`	
`f64.const`	0x44	`$value` : float64	`() : (f64)`	

Variables Access

- Get Local (id is the stack slot - push on stack)

Mnemonic	Opcode	Immediates	Signature	Families
`get_local`	0x20	`$id` : varuint32	`() : ($T[1])`	

- Set Local (id is the stack slot - pop from stack)

Mnemonic	Opcode	Immediates	Signature	Families
`set_local`	0x21	`$id` : varuint32	`($T[1]) : ()`	

- Get Global (id in the global index space - push on stack)

Mnemonic	Opcode	Immediates	Signature	Families
`get_global`	0x23	`$id` : varuint32	`() : ($T[1])`	

- Set Global (id in the global index space - pop from stack)

Mnemonic	Opcode	Immediates	Signature	Families
`set_global`	0x24	`$id` : varuint32	`($T[1]) : ()`	

JWebAssembly load and store are always local

Conversion Instructions

- Truncate Floating-Point to Integer, Signed

Mnemonic	Opcode	Signature	Families
`i32.trunc_s/f32`	0xa8	`(f32) : (i32)`	F, S
`i32.trunc_s/f64`	0xaa	`(f64) : (i32)`	F, S
`i64.trunc_s/f32`	0xae	`(f32) : (i64)`	F, S
`i64.trunc_s/f64`	0xb0	`(f64) : (i64)`	F, S

- Round Floating-Point to nearest Integer, Signed; but type is still Floating Point!

Mnemonic	Opcode	Signature	Families
`f32.nearest`	0x90	`(f32) : (f32)`	F
`f64.nearest`	0x9e	`(f64) : (f64)`	F

… Conversion Instructions

- Convert Integer To Floating-Point and back, Signed

Mnemonic	Opcode	Signature	Families
`f32.convert_u/i32`	0xb3	`(i32) : (f32)`	F, U
`f32.convert_u/i64`	0xb5	`(i64) : (f32)`	F, U
`f64.convert_u/i32`	0xb8	`(i32) : (f64)`	F, U
`f64.convert_u/i64`	0xba	`(i64) : (f64)`	F, U

- other Conversions ….

- Remark: in JWebAssembly the classes **`WasmConvertInstruction & ValueTypeConvertion`** use other nonstandard Mnemonics: **i2d, d2i,** etc.

Arithmetic Instructions

- Arithmetic Instructions for the 4 types
  - take two operands of given type from estack and places result on estack

i32.add	i64.add	f32.add	f64.add
i32.sub	i64.sub	f32.sub	f64.sub
i32.mul	i64.mul	f32.mul	f64.mul
i32.div_s i32.div_u	i64.div_s i64.div_u	f32.div	f64.div

Integer Comparison Instructions

- **eq** Integer Equality

Mnemonic	Opcode	Signature	Families
`i32.eq`	0x46	`(i32, i32) : (i32)`	C, G
`i64.eq`	0x51	`(i64, i64) : (i32)`	C, G

- Result as i32 on the stack (boolean interpretation)
- **eqz** tests whether the operand is zero
- **ne** tests whether the operands are not equal.
- **lt_s** signed test whether the first operand is less than the second operand
- **lt_u** unsigned test whether the first operand is less than the second operand
- **le_s** signed test whether the first operand is less than or equal to the second
- **le_u** unsigned test whether the first operand is less than or equal to the second
- **gt_s**, **gt_u**, **ge_s**, **ge_u** accordingly

Floating-Point Comparison Instructions

- **eq** Floating Point Equality

Mnemonic	Opcode	Signature	Families
`f32.eq`	0x5b	`(f32, f32) : (i32)`	C, F
`f64.eq`	0x61	`(f64, f64) : (i32)`	C, F

- Result as i32 on the stack (boolean interpretation)

- **ne** tests whether the operands are not equal.
- **lt** tests whether the first operand is less than the second operand
- **le** tests whether the first operand is less than or equal to the second
- **gt**, **ge** accordingly

IF ELSE END

- **if** tests if value t.o.s is 0 or value != 0
  - if value != 0 the following instructions are executed (ending with end or else)
- **else** instructions of if had value 0
- **end** terminates the if (and else)

```
get_local 0
i32.const 0
i32.eq
if  ;; label = @1
  i32.const 32
  set_local 0
else
  i32.const 44
  set_local 0
end
```

BLOCK and LOOP

- **block** and **end** a nested label is defined at its **end**
- **loop** and **end** a label is defines at its **beginning** (but no automatic loop execution!, needs a br 0 before end)

- **br <label>** branch unconditionally to label (value is nesting level)
- **br_if <label>** branch if value top of stack is != 0
  - the labels are automatically nested
  - i.e. br 0 branches to innermost label

```
void print_str(i32 addr) {
  i32 byte = 0;
  while (true) {
    byte = i32.load8_u(addr);
    if (byte == 0) { break; }
    putc(byte);
    addr = addr + 1;
    continue;
  }
}
```

```
block void          // declares a "label" at it's "end"
loop label 0        // declares a "label" right here
    // byte = i32.load8_u(addr)
    get_local 0     // push addr
    i32.load8_u 0 0 // push the byte at addr as an i32
    tee_local 1     // store the byte to local 1, but don't pop it

    // if (byte == 0) { break }
    i32.eqz         // (x i32) => i32(x == 0 ? 1 : 0)
    br_if 1         // if the byte was zero, jump to end of "block"

    // putc(byte)
    get_local 1     // push byte
    call 0          // call imported "putc" function with the byte

    // addr = addr + 1
    get_local 0     // push addr
    i32.const 1     // push i32 "1"
    i32.add         // push result from addr + 1
    set_local 0     // store new address to "addr" local

    // continue
    br 0            // jump to "loop" (i.e. continue looping)
  end
end // label 1: "block"
```

CALL and RETURN

Call a Function

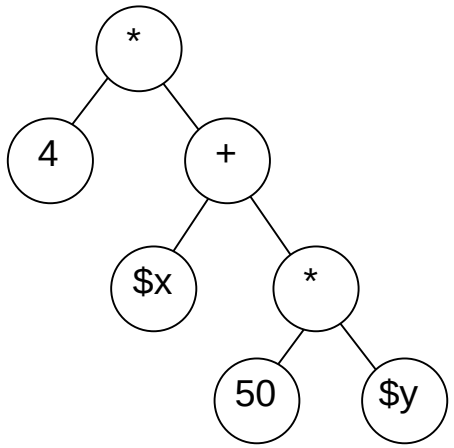
- Arguments pushed to expression stack are (virtually) transferred to call stack

Return

- Value of expression stack is (virtually) transferred to call stack

Abstract Syntax Tree Notation

- WASM are also representing an AST
  - binary format can easily be converted (back) into it
- Notation are so called s-expressions (defined by LISP as "prefix notation")




```
(i32.mul
  (i32.add
    (i32.mul
      (i32.const 50)
      (get_local $y)
    )
    (get_local $x)
  )
  (i32.const 4)
)
```

```
c:> wasm2wat   --fold-exprs   TestAdd.wasm
```

https://webassembly.github.io/wabt/demo/wat2wasm/

LISP: Language with Prefix Notation

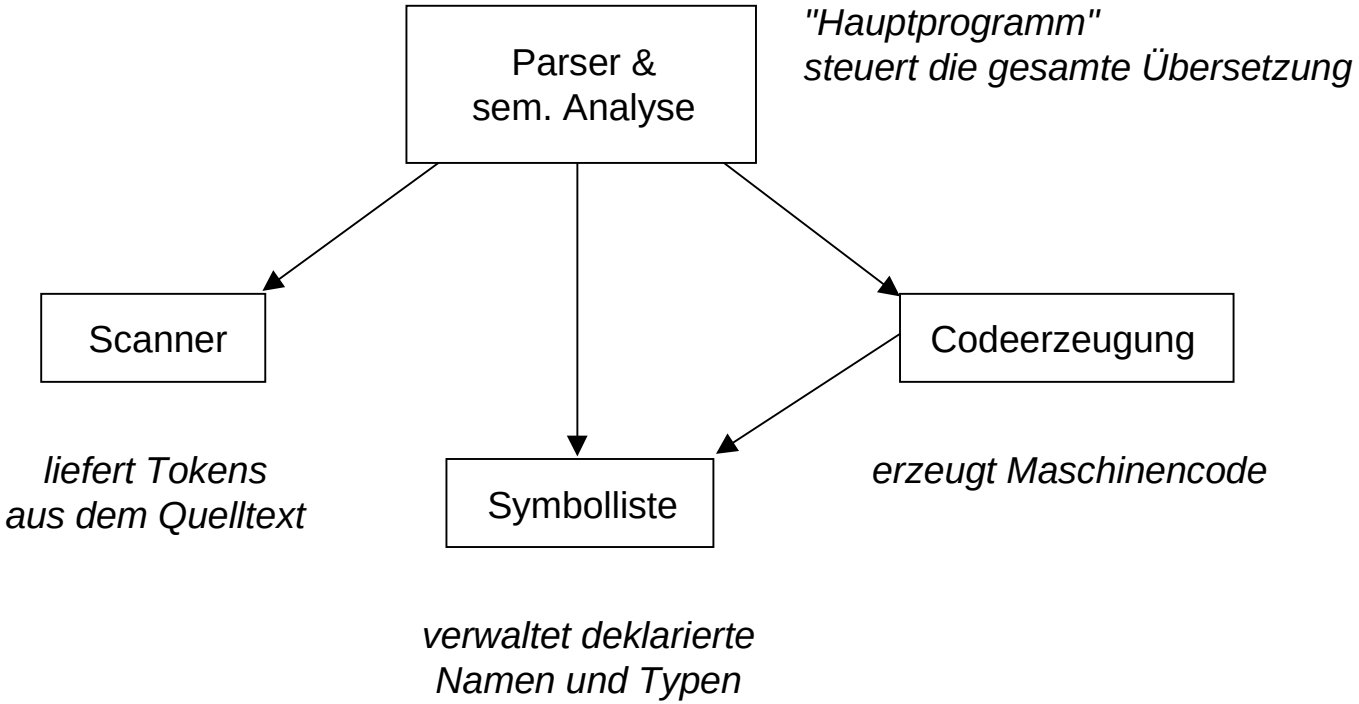
School of Engineering

- 1958 – Pioneering Language
- Treats all operators as user-defined ones
  - syntax does not assume the number of arguments is known
  - use parantheses in prefix notation: write f(x,y) as (f x y)

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Zusammenfassung

- Wesentlichen Teile eines Compilers



Noch Fragen?

Anhang Lösung: Arbeitsweise eines PDA

Keller	*restliche Eingabe*
0 .	((x))
0 2 .	(x))
0 2 2 .	x))
0 2 2 1 .	))
0 2 2 .	E))
0 2 2 3 .	))
0 2 2 3 4 .	)
0 2 .	E)
0 2 3 .	)
0 2 3 4 .	
0 .	E

Appendix: WebAssembly

- Overview of bytecodes:
  http://webassembly.org/docs/semantics/
  https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md
- Compiling from C:
  http://webassembly.org/getting-started/developers-guide/
  https://hacks.mozilla.org/2017/03/previewing-the-webassembly-explorer/
- Introduction to WebAssembly:
  https://rsms.me/wasm-intro
- Paper about WebAssembly:
  https://www.cs.tufts.edu/~nr/cs257/archive/andreas-rossberg/webassembly.pdf

Appendix: JWebAssembly

- Open Source project to convert Java Programs to WASM
- Only static methods with primitive types supported - yet
- Also provided with with an interface to emit code directly

Signatur of function

code emitter

```
JWebAssembly.emitCode(Class signature, Emitter codegen)
```

- **JWebAssembly.il** instruction List
- int **JWebAssembly.**local(<type>,<name>) get local slot number of variables
  - $arg0, … $argn predefined names of parameters
- Instructions are defined in the following classes

```
WasmConstInstruction
WasmLoadStoreInstruction // always local
WasmBlockInstruction & WasmBlockOperator
WasmNumericInstruction & NumericOperator
WasmConvertInstruction & ValueTypeConvertion
WasmCallInstruction
```

Appendix: JWebAssembly Sample

```
import de.inetsoftware.jwebassembly.*;
import de.inetsoftware.jwebassembly.module.*;

public class TestAdderEmitter implements Emitter{
     interface TestAdder {
        int add();
     }

     public void emit() {
        // Instruction Example: 5 + 8
        // load int const 5 on stack
        JWebAssembly.il.add(new WasmConstInstruction(5,0));

        // load int const 8 on stack
        JWebAssembly.il.add(new WasmConstInstruction(8,0));

        // add it
        JWebAssembly.il.add(new WasmNumericInstruction(NumericOperator.add,ValueType.i32,0));
        // result top of stack
        JWebAssembly.il.add(new WasmBlockInstruction(WasmBlockOperator.RETURN, null, 0 ));
     }

     public static void main(String[] args) throws Exception {
        JWebAssembly.emitCode(TestAdder.class, new TestAdderEmitter());
     }
}
```

Appendix: … JWebAssembly Sample

```
import de.inetsoftware.jwebassembly.*;
import de.inetsoftware.jwebassembly.module.*;

public class TestAdder2Emitter implements Emitter{

     interface TestAdder2 {
        double add(double a, double b);
     }
                                              }
     public void emit() {
        // 1 + arg0 + arg1 as double
        // load int const 1 on stack
        JWebAssembly.il.add(new WasmConstInstruction(1.0,0));
        JWebAssembly.il.add(new WasmLoadStoreInstruction( true,
           JWebAssembly.local(ValueType.f64,"$arg0"), 0 ));
        // add it
        JWebAssembly.il.add(new WasmNumericInstruction(NumericOperator.add,ValueType.f64,0));
        JWebAssembly.il.add(new WasmLoadStoreInstruction( true,
           JWebAssembly.local(ValueType.f64,"$arg1"), 0 ));
        // add it
        JWebAssembly.il.add(new WasmNumericInstruction(NumericOperator.add,ValueType.f64,0));
        // result top of stack
        JWebAssembly.il.add(new WasmBlockInstruction(WasmBlockOperator.RETURN, null, 0 ));
     }

     public static void main(String[] args) throws Exception {
        JWebAssembly.emitCode(TestAdder2.class, new TestAdder2Emitter());
     }
}
```

Appendix: JWebAssembly java -> wasm

- also: Java class Files converted to wasm
  - restricted to primitive types and static methods
  - Exported methods need to have @Export annotation
- java -jar JWebAssembly.jar Factorial.class

```
package ch.zhaw.psp;

import de.inetsoftware.jwebassembly.api.annotation.Export;
import de.inetsoftware.jwebassembly.api.annotation.Import;
import de.inetsoftware.jwebassembly.*;

public class Factorial {

     @Export
     static int factorial(int n) {
       if (n == 0)
         return 1;
       else
         return n * factorial(n-1);
     }

}
```

Appendix: WASM Utilities

- wasm2wat
  - parse binary file wasm-file and write wat text file
- wasm-objdump
  - print information about the contents of wasm binaries.
- wasm-interpret
  - read a file in the wasm binary format, and run in it a stack-based interprete

use this one if wasm2wat fails

Additional Tools

- wasm-validate
  - read a file in the WebAssembly binary format, and validate it.
- wat2wasm
  - parse wat-file and write to binary file wasm-file
- wat-desugar
  - read a file in the wasm s-expression format and format it.
- wasm2c
  - read a file in the WebAssembly binary format, and convert it to a C source file and header.

Appendix: All WASM Instructions

School of Engineering

i32.add	i64.add	f32.add	f64.add	i32.wrap/i64	i32.load8_s	i32.store8
i32.sub	i64.sub	f32.sub	f64.sub	i32.trunc_s/f32	i32.load8_u	i32.store16
i32.mul	i64.mul	f32.mul	f64.mul	i32.trunc_s/f64	i32.load16_s	i32.store
i32.div_s	i64.div_s	f32.div	f64.div	i32.trunc_u/f32	i32.load16_u	i64.store8
i32.div_u	i64.div_u	f32.abs	f64.abs	i32.trunc_u/f64	i32.load	i64.store16
i32.rem_s	i64.rem_s	f32.neg	f64.neg	i32.reinterpret/f32	i64.load8_s	i64.store32
i32.rem_u	i64.rem_u	f32.copysign	f64.copysign	i64.extend_s/i32	i64.load8_u	i64.store
i32.and	i64.and	f32.ceil	f64.ceil	i64.extend_u/i32	i64.load16_s	f32.store
i32.or	i64.or	f32.floor	f64.floor	i64.trunc_s/f32	i64.load16_u	f64.store
i32.xor	i64.xor	f32.trunc	f64.trunc	i64.trunc_s/f64	i64.load32_s	
i32.shl	i64.shl	f32.nearest	f64.nearest	i64.trunc_u/f32	i64.load32_u	
i32.shr_u	i64.shr_u			i64.trunc_u/f64	i64.load	call
i32.shr_s	i64.shr_s	f32.sqrt	f64.sqrt	i64.reinterpret/f64	f32.load	call_indirect
i32.rotl	i64.rotl	f32.min	f64.min		f64.load	
i32.rotr	i64.rotr	f32.max	f64.max			
i32.clz	i64.clz				nop	grow_memory
i32.ctz	i64.ctz				block	current_memory
				f32.demote/f64	loop	
i32.eqz	i64.eqz			f32.convert_s/i32	if	get_local
i32.eq	i64.eq			f32.convert_s/i64	else	set_local
i32.ne	i64.ne			f32.convert_u/i32	br	tee_local
i32.lt_s	i64.lt_s			f32.convert_u/i64	br_if	
i32.le_s	i64.le_s			f32.reinterpret/i32	br_table	get_global
i32.lt_u	i64.lt_u	f32.eq	f64.eq	f64.promote/f32	return	set_global
i32.le_u	i64.le_u	f32.ne	f64.ne	f64.convert_s/i32	end	
i32.gt_s	i64.gt_s	f32.lt	f64.lt	f64.convert_s/i64		i32.const
i32.ge_s	i64.ge_s	f32.le	f64.le	f64.convert_u/i32	drop	i64.const
i32.gt_u	i64.gt_u	f32.gt	f64.gt	f64.convert_u/i64	select	f32.const
i32.ge_u	i64.ge_u	f32.ge	f64.ge	f64.reinterpret/i64	unreachable	f64.const