Shift-reduce parser

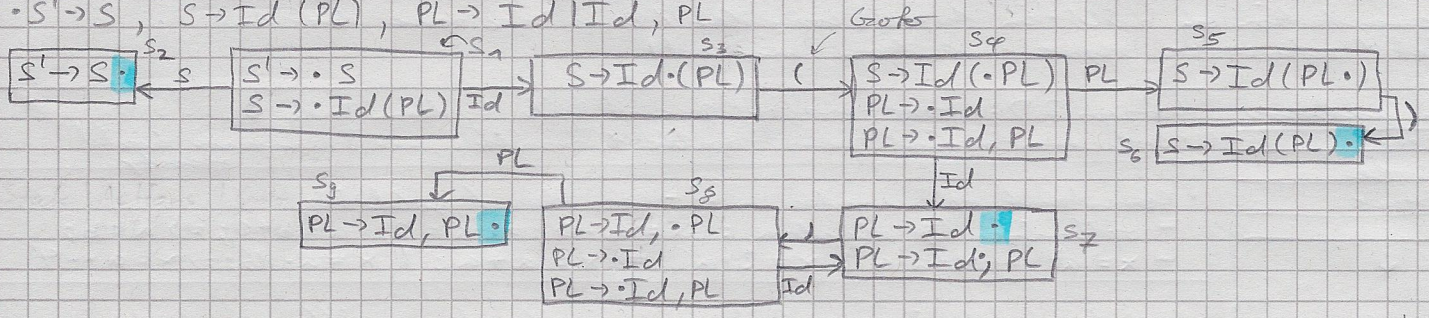
- "s2": next char on stack, state 2 on stack
- "r5": reduce with Rule 5, next state: last state on stack

LL(1) parsing table

- Zeilen: Zustandsnummer, Spalten: Terminale (nächste "Aktion" (r?, s?) und nicht-Terminale ein Goto Bereich
- $ nicht vergessen! ①
- Ergänze: Falls 'S' Start ist, füge S' → S hinzu

SLR

- S' → S (1), S → Id (PL) (2), PL → Id | Id, PL (3)



- falls "A → α · a β" in S_i und Goto(S_i, a) = S_j ⇒ M[S_i, a] = shift j
- falls "A → α ·" in S_i und t ∈ Follow(A) ⇒ reduce with rule "A → α" in M[S_i, t]
- falls "S' → S ·" in S_i ⇒ M[S_i, $] = accept
- alles andere "Fehler" (leerer Eintrag)
- Follow(A): alle Terminale x, für die das Wort α A x β ergänzt werden können laut Grammatik von S → ... . Also alle Terminale, die auf das nicht-Terminal A folgen dürfen.

	Aktions					Goto	
	Id	(	)	,	$	S	PL
1	s3					2	
2					acc		
3		s4					
4	s7					5	
5			s6				
6				r2			
7			r3	s8			
8	s7					9	
9			r3				

- Goto(S_i, Z) = S_j Z ∈ Terminal or Nichtterminal definiert als: für alle "X → α · Z β" in S_i ⇒ dann füge "X → α Z · β" zu S_j hinzu
- Closure(S_k) = S_k ∪ {alle Items "B → · γ" für jedes "X → α · B β", B nicht-Terminal, "B → γ" in Regeln}
- M[top of stack (state), next symbol] ←
- bottom-up parser

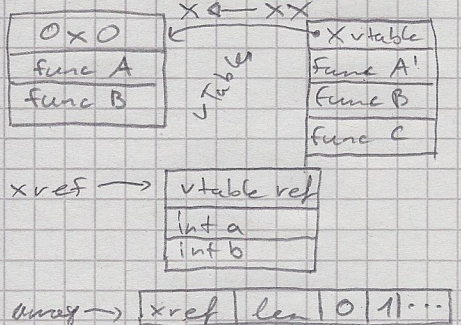
A: Compiler can always check
S: Sometimes compiler can check
D: compiler has to generate code to check
X: impossible

- X reaches the end of program
- A,S,D program computes only values that can be represented (overflow)
- D contracts
- X/A program ends before timelimit
- S no null references used for object access

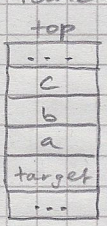
- A Syntax
- A Names/Symbols well defined
- S Variables are initialized before reading
- A/D rules of typesystem obeyed
- A all possible exceptions are caught
- A functions are defined
- S all code reachable (no dead code)
- A no calls/access to private members
- X program does not exhaust memory
- X powerconsumption with limits

Register algo
- Alle BinOps/UnOps brauche und notiere, wie viele Register sie brauche.
- Beim Code generieren priorisiert die Seite mit mehr Registern

- L = {w c w | w ∈ {a,b}*} Σ = {a,b,c} ⇒ any variable used after a marker "c" has been defined before. not context free
- L = {$a^n b^m c^n d^m$} n,m ≥ 0 ⇒ for any function the number of formal parameters is equal to the number of actual parameters (compare declaration with call) → not context free



func(a,b,c)
⇒ func(target, a, b, c)



id (id)

stack	to read	action
$$S_1$	id(id)$	S_3
$$S_1 id S_3$	(id)$	S_4
$$S_1 id S_3 (S_4$	id)$	S_7
$$S_1 id S_3 (S_4 id S_7$	)$	r_3
$$S_1 id S_3 (S_4 PL S_5$	)$	S_6
$$S_1 id S_3 (S_4 PL S_5)$	$	r_3
$$S_1 S_2$	$	acc ✓

by value (copy), by ref (reference mem), by result, by name (substitution)
↳ f(B[0])
↳ tmp = B[0];
// use tmp for para
B[0] = tmp
→ multiple return values (C# "out")

Graph coloring

remove nodes with $< k$ neighbors and their edges. If no nodes left (empty graph) $\Rightarrow$ graph is k-colorable otherwise (nodes left) not.

Top-down parser

- First(α): for a word α (with terminals and non-terminals) we define:
  $First_1(\alpha) = \{t \mid t$ is a terminal and there is a derivation $\alpha \Rightarrow \dots \Rightarrow t\beta\}$
  $First_k(\alpha) = \{s \mid s$ word of terminals, $|s| \leq k$, such that $\alpha \Rightarrow \dots \Rightarrow s\beta\}$

- First(X) Algorithm: do until no more can be added to First(X) (terminal or ε). X is terminal or non-terminal.
  (i) if X is terminal $\Rightarrow$ First(X) = {X}
  (ii) if "$X \to \varepsilon$" is a production $\Rightarrow$ add ε to First(X)
  (iii) if X non-terminal and $X \to Y_1 Y_2 Y_3 \dots Y_n$ a production
  $\Rightarrow$ if 'a' $\in$ First(Y_1) $\Rightarrow$ add a to First(X)
  if $\varepsilon \in$ First(Y_1) and $\varepsilon \in$ First(Y_2) and ...
  ... and $\varepsilon \in$ First(Y_k) (Vorsicht $\varepsilon \notin$ First(Y_{k+1}) (iii) nicht erwartet)
  then add First(Y_{k+1}) to First(X)

- Follow(X): set of terminals 'a' such that there is a derivation $S \Rightarrow \alpha X a \beta$ for some α, β words (non-terminals and terminals)
  - Follow(X) algo: Do until nothing can be added:
    (i) put $ into Follow(S), S is start!
    (ii) for '$A \to \alpha B \beta$' rules: First(β) $\leq$ Follow(B) except ε!
    (iii) for '$A \to \alpha B$' or '$A \to \alpha B \beta$' with $\beta \Rightarrow \dots \Rightarrow \varepsilon$ ($\varepsilon \in$ First(β)) added Follow(A) to Follow(B)

$\Rightarrow$ Konstruktion Table M for top-down parser benötigt First() und Follow():
For all '$A \to \alpha$' we do the following: $\in \alpha = \alpha'\beta$, nur α' (1 Symbol) ansehen
(i) $\forall a \in$ First(α) add '$A \to \alpha$' to $M[A, a]$
(ii) if $\varepsilon \in$ First(α) add '$A \to \alpha$' for all $b \in$ Follow(A) to $M[A, b]$
(iii) $\hookrightarrow$ also if b == $

Rules: $E \to T\,E'$, $E' \to +T\,E' \mid \varepsilon$, $T \to F\,T'$, $T' \to *F\,T' \mid \varepsilon$, $F \to (E) \mid id$

$\Rightarrow$ First Table (transponiert)

	id	+	*	(	)	E	E'	T	T'	F
rule (i)	{id}	{+}	{*}	{(}	{)}					
rule (ii)							{ε}		{ε}	
rule (iii), first 'if'						{ε, +}		{(, id}		{(, id}
" " "									{ε, *}	
" " "						{(, id}		{(, id}		
result	{id}	{+}	{*}	{(}	{)}	{(, id}	{ε, +}	{(, id}	{ε, *}	{(, id}

$\Rightarrow$ Follow Table (transponiert)

	E	E'	T	T'	F
rule (i)	{$}				
rule (ii)	{$,)} ↩ First()		First(E') ↓ {+}		First(T') ↓ {*}
rule (iii), with no β		{$,)} ↙ Follow(E)	↳ Follow(E) ↳ {$,), +}	Follow(T) ↓ {+}	↳ Follow(T) (and Follow(T')) {+, *, $,)}
rule (iii), with $\beta \Rightarrow \dots \Rightarrow \varepsilon$			→ also Follow(E')	{+, $,)} ↖ Follow(T), weil $T \to FT'$	
result:	{$,)}	{$,)}	{$,), +}	{+, $,)}	{+, *, $,), }

$\Rightarrow$ Table

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to Id$			$F \to (E)$		

Top-Down Schritte:
if (top of Stack is terminal) {
  if (top is same as next char) { pop Stack; pop next char }
  else { error }
} else if (top is non-Terminal) { if M[top, next char] = $X \to Y_1 \dots Y_n$ {
  pop Stack, push $Y_1 \dots Y_n$ } else { error (M[..., ...] empty) }
} else if (top is $ and next char is $) { accept }
else { error }

... Example $w = Id + Id * Id$

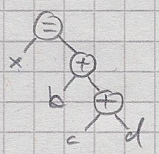
Stack (stack top ↓, start: E $, stack "bottom")	w left	comment
E $	Id + Id * Id $	M[E, Id]
T E' $	Id + Id * Id $	M[T, Id]
F T' E' $	Id + Id * Id $	M[F, Id]
id T' E' $	Id + Id * Id $	Id == Id : pop
T' E' $	+ Id * Id $	M[E', +]
...	...	...

constant folding:
- T ≡ var is not constant
- ⊥ ≡ stmt is, as far we know, not executed
- {var = v} ≡ var is constant v

	T	⊥	v
T	T	T	T
⊥	T	⊥	v: from above v
v	T	v	v, if both same value

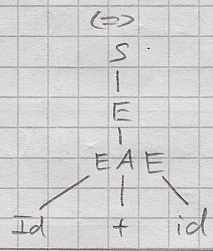
- constant propagation: "and", top down
- common subexpression elimination: "and", top down
- busy expression: "and", bottom up
- live variables: "or", bottom up
- reaching definitions: "or", top down

$x = b + c + d$



- context-free ⇔ $\mathcal{L}_2$ (≡ Type 2 grammer (≡) push-down automaton)
- linear Grammer: $X \to P_1 \vee P_2$ (Left Grammer / right Grammer)
- linear Grammer ⇔ Type 3 grammer ⇔ regular grammer (⇔ regular exp.)

- $S \to E \to EAE \to EA\,id \to E + id \to id + id$



Grammer G is ambigious if one of the following hold:
- there are multiple parse trees for some $w \in L(G)$
- there are multiple left most (right most) derivations for some $w \in L(G)$

source ⟷ Finite State Machine Lexer | Tokens → parser ↖ stack-machine

"maximum munch" (lexer): read chars, as long as it belongs to the Token (valid char) or until white space found. a ? y : y. Works not always (stop stop) "pointer magic C"

Backtracking: try first rule, if fails, return to last decision with possible rules left and try next. Repeat until all possible decisions made or success. Works always ⇒ $O(n^2)$ or worse?

If Grammer not LL(1): - rewrite Rules ($X \to cC$, $C \to cC \mid a \mid b$)
- increase k in LL(k) (not liked) - bottom up parsing (preferred)

Symbol Table (Semantic Check) („ST")

global ST.

name	type
A	class
B	class
int	...
bool	...
...	
true	bool
false	bool

A ST

name	type
a	int
b	float
foo	method ret: void

A.foo ST

i	param, int
a	int

```
class A {
  int a;
  float b;
  void foo(int i) {
    int a;
  }
}
```

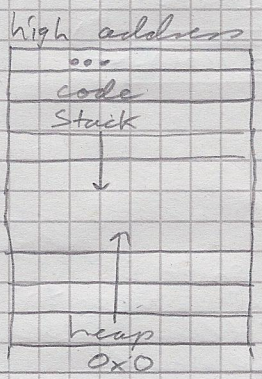
+ offset
+ visibility
+ ...

Checks (e.g. Array access)
- „explicit": add check-code directly to AST ⇒ messy
- „delayed handling": Semantic analyzer marks nodes to be checked, generate check-code at the last moment (code generation)

↙ "RTS"

Runtime System: • interacts with OS • provides services to compiler • compiler assumes existence of runtime system • compiler interacts with runtime system (GC: save points to run GC; GC root set) • RTS defines how memory is used • RTS contains a "loader" (load program and start it) • RTS defines how processes interact

RTS ↪ Compiler must agree on: • Process layout (driven by RTS) • representation of built-in types (compiler / language spec driven) • Object / reference representation (mixed)

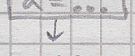


Arrays 2D: • row-major: zuerst Zeile 1, dann Zeile 2, ... • column-major: zuerst Spalte 1, Spalte 2, ... | language should specify

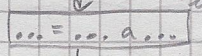
• Constant optimization: - constant folding
- optimize by algebra: $a = b \cdot 0; \Rightarrow a = 0;$ $a = b + 0; \Rightarrow a = b;$
- $x = a^2 \Rightarrow x = a \cdot a$ | $x = 2 \cdot a; \Rightarrow x = a + a;$ | $x = a \cdot 16; \Rightarrow x = a << 4$

Constant folding and loops: initialize with $\perp$, loop until nothing changes. Will stop ("converge")

Live variables: An assign statement is dead, if the variable is not used after the statement. Bottom-up analysis.



in: dead | dead
out: lives | dead



in: lives | lives
out: lives | dead

read

merge: lives + lives = lives, dead + lives = lives, dead + dead = dead
loops: until no change!

Variables initialized: assume no var is initialized. Go from top to bottom and change to "initialized" if assignment happens to the variable. Merge: initialized + initialized = initialized, otherwise not. (not perfect analysis!)

Register interference graph („Rig"): nodes ≙ vars, edge ≙ need to be in register at the same time. Create "live lines" (first definition - last usage) → add edges for "overlaping" lines. Add edge if var in same statement.