Compiler principles

Compiler

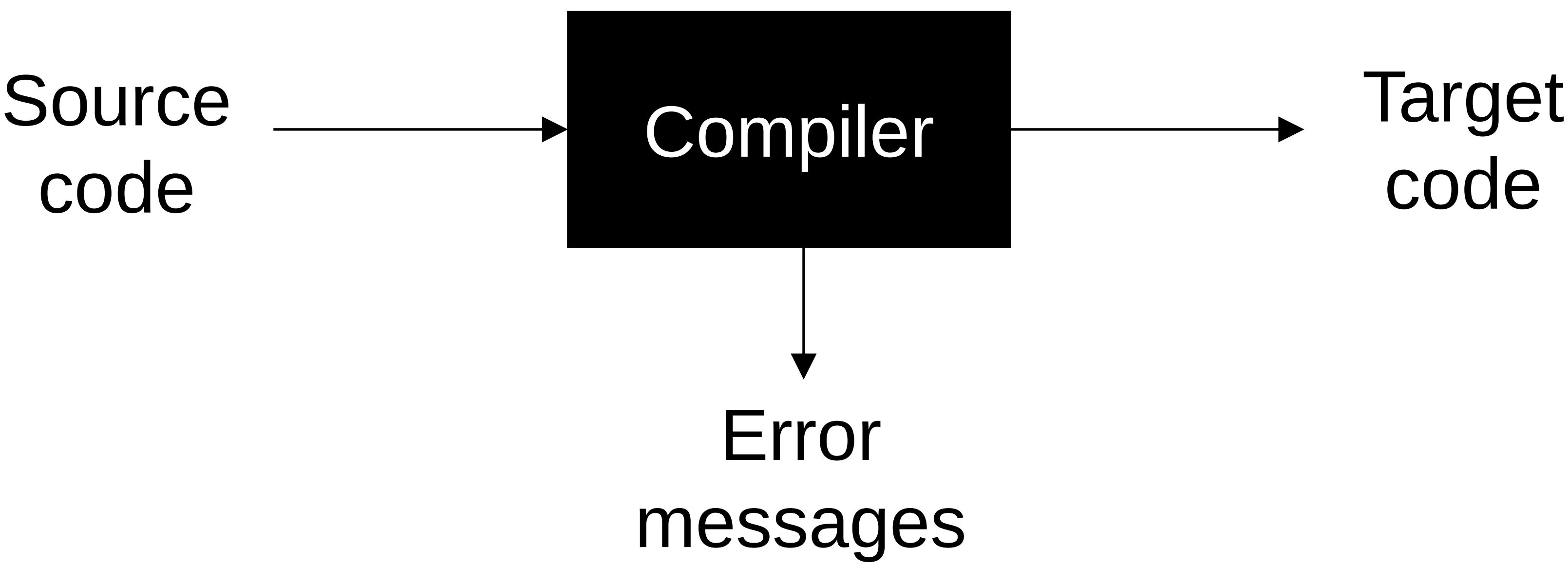
Jakub Yaghob

Literature and slides

- The Dragon Book
  - Aho, Sethi, Ullman: Compilers - Principles, Techniques and Tools, Addison-Wesley 1986
  - Aho, Lam, Sethi, Ullman: Compilers - Principles, Techniques and Tools (2nd edition), Addison-Wesley 2006
- Advanced compiler techniques
  - Muchnick S.S.: Advanced compiler design and implementation, Morgan Kaufman Publishers 1997
- Slides
  - http://www.ksi.mff.cuni.cz/lectures/NSWI098/html/index.html

What is a compiler?

- Naïve concept
  - A black-box compiling a source code to a target code



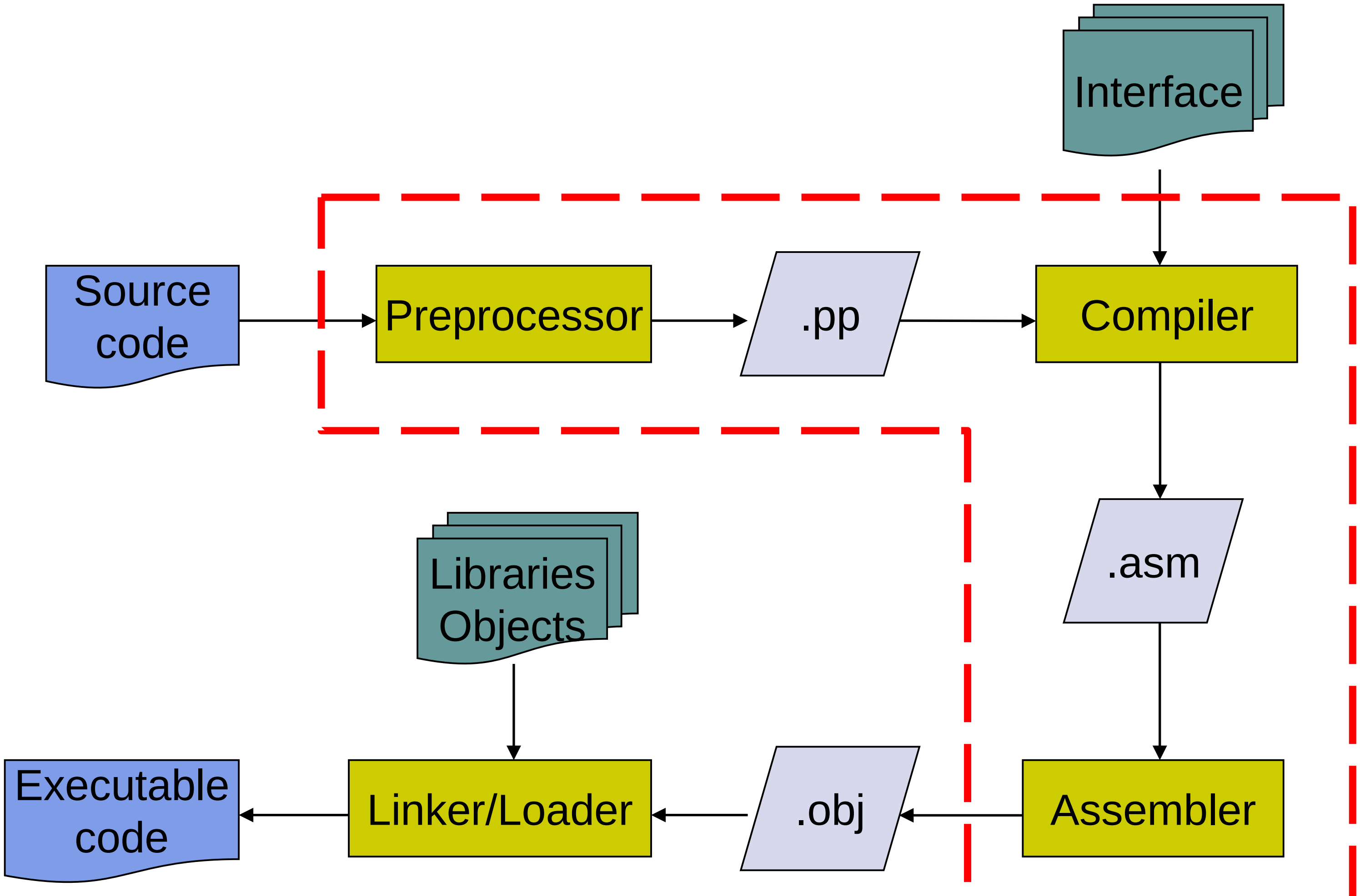
What is a compiler? More formally

- Let's have an input language L_{in} generated by a grammar G_{in}
- Let's have an output language L_{out} generated by a grammar G_{out} or accepted by an automaton A_{out}
- The compiler is a mapping $L_{in} \rightarrow L_{out}$, where $\forall w_{in} \in L_{in} \; \exists \; w_{out} \in L_{out}$. The mapping does not exist for $w_{in} \notin L_{in}$

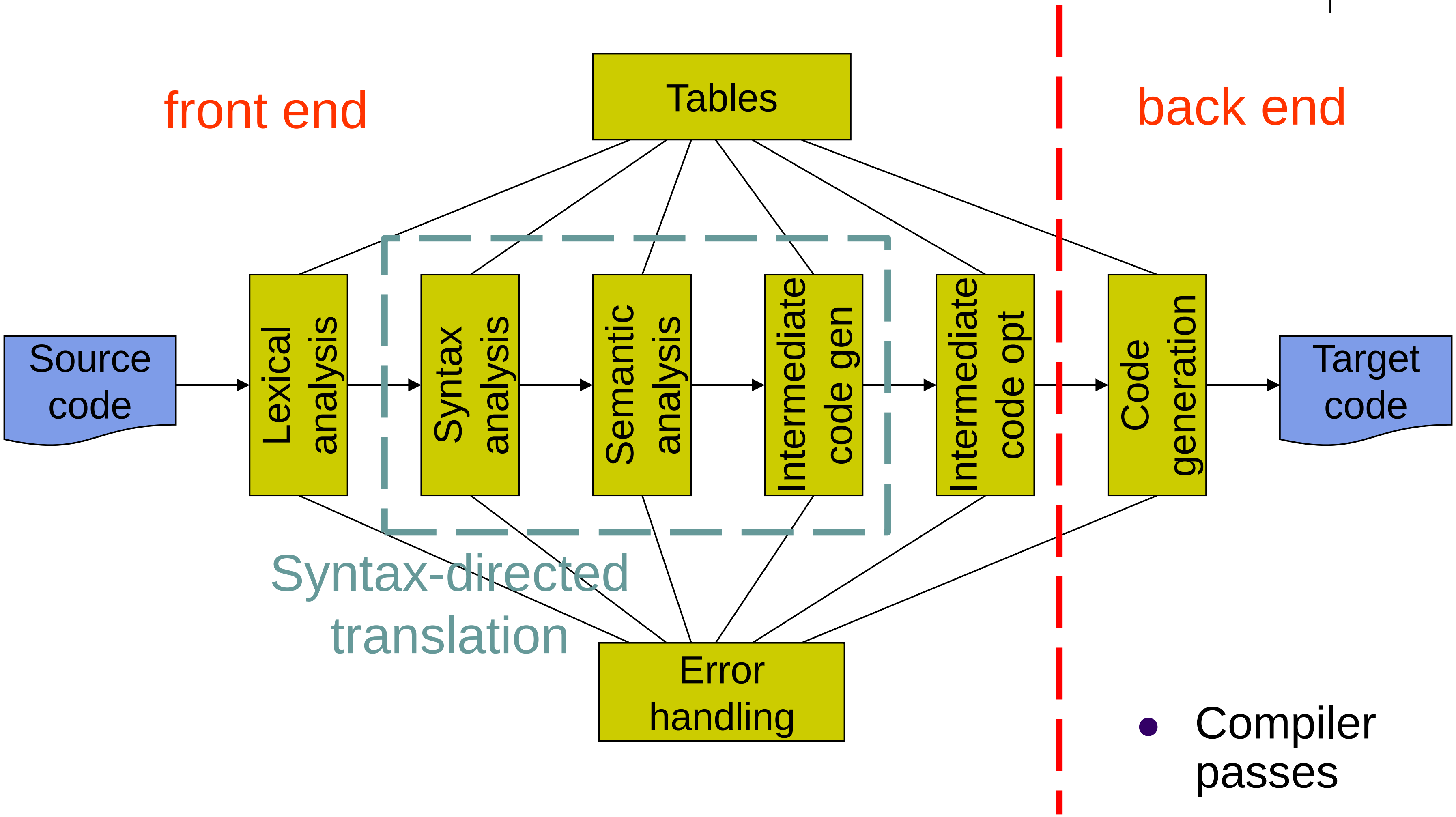
Use cases

- Structured (e.g. MPS from JetBrains) or syntax-highlighting editor
- Pretty-printer
- Static program checker
  - LINT
- Interpreters
- Modelling languages compiler
  - Verilog, VHDL
- Query languages
  - SQL

Program translation



Phases of a compiler



- Compiler passes

Compiler-construction tools

- Parser generators
  - Produce syntax analyzers
  - Usually description based on a context-free grammar
  - Bison, Coco/R, ANTLR
- Scanner generators
  - Produce lexical analyzers
  - Usually description based on regular expressions
  - Flex
- Automatic code generators
  - Produce translations for each intermediate code instructions to the target code
  - A processor model and description
  - Mono JIT

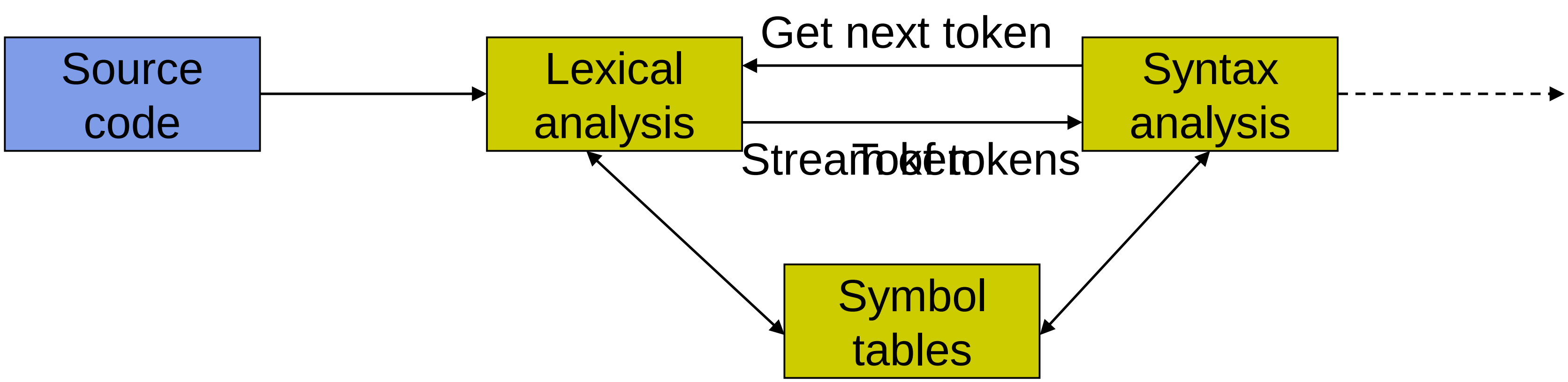
Compiler principles

Lexical analysis

Jakub Yaghob

Lexical analysis

- Reads input characters and produces a sequence of tokens
- Simpler design
  - Separation of lexical analysis from syntax analysis
- Specialization
  - Speedup
- Enhanced portability
  - Moving a compiler to other platform requires changes only in lexical analysis (EBCDIC)
- „Increasing“ grammar look-ahead
  - LR(1) does not mean look-ahead for 1 character
- Support for macros usually in lexical analysis



Lexical analysis terms

- Token
  - Lexical analysis output and syntax analysis input
  - Called terminal on the syntax analysis/grammar side
  - A set of strings producing the same token
- Pattern
  - Rules describing a set of strings for given token
  - Usually described by regular expressions
- Lexical element
  - A sequence of characters in a source code corresponding with a pattern of a token
  - Some lexical elements do not have corresponding token
    - Comment
- Literal
  - A constant, has a value

Examples

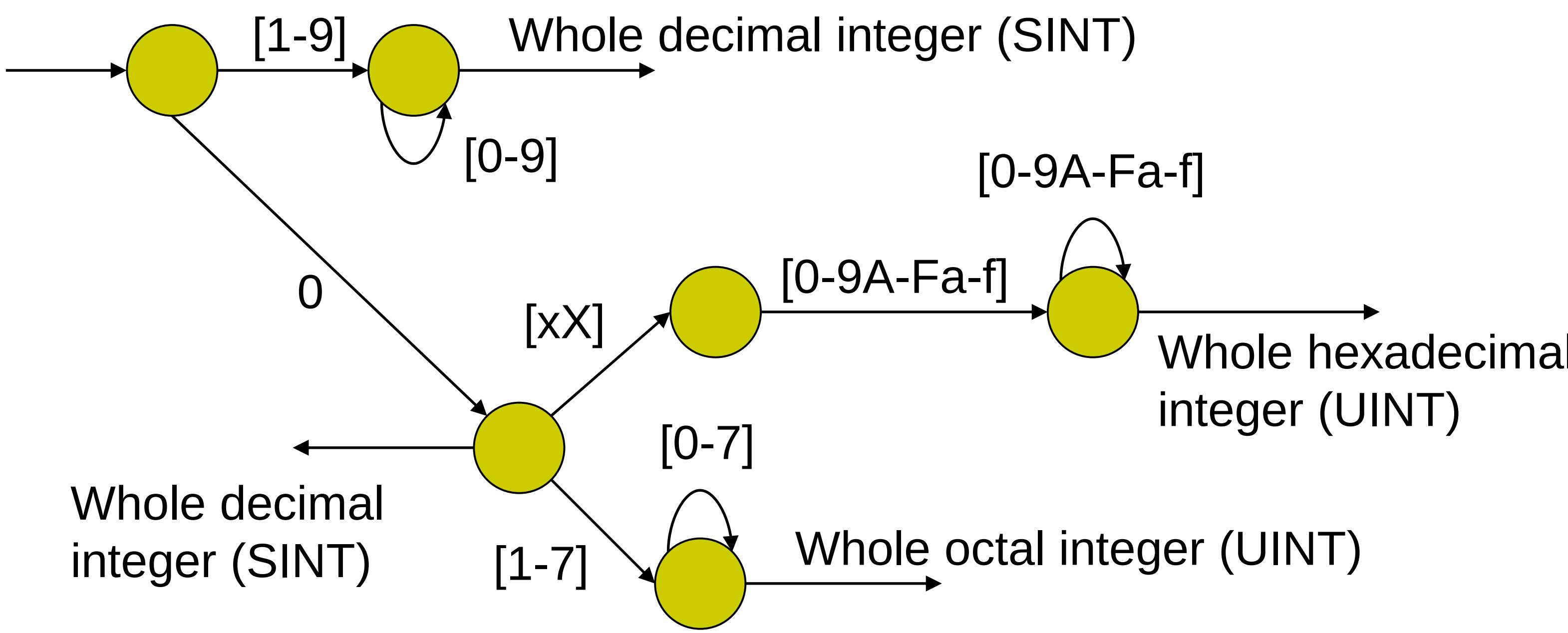
<table>
<tr><th>Token</th><th>Lexical element</th><th>Regular expression</th></tr>
<tr><td>while</td><td>while</td><td>while</td></tr>
<tr><td>relop</td><td><,<=,=,<>,>,>=</td><td>\<|\<=|=|\<>|>|>=</td></tr>
<tr><td>uint</td><td>0, 123</td><td>[0-9]+</td></tr>
<tr><td></td><td>/* comment */</td><td>\/* → cmt, <cmt>., <cmt>*\/</td></tr>
</table>

Interesting problems in lexical analysis

- Line indentation
  - Some languages have the indentation as a syntax construction
    - Python, Flex
- Identifiers
  - Identifiers with spaces
    - DO 5 I = 1.25
    - DO 5 I = 1,25
  - Keywords as identifiers
- Context dependent tokens
  - The value of a token depends on some other information
    - a*b;

Lexical analysis background

- Patterns using regular expressions → regular languages → accepted by finite automata
- Restarting the automaton after each recognized/accepted token
- Finite automaton for an integer in C:



Token attributes

- More patterns recognized as a token or the token is a literal
- Usually one attribute specifying more precisely a token or a literal value
  - Token=relop, specification=‘<=’
  - Token=uint, specification=‘123’

Lexical errors

- The finite automaton cannot continue and it is not in a final state
  - Unknown character
  - Unfinished string at the end of line
- Recovery
  - Ignore it
  - Deduce missing character(s)
- Typo in a keyword is not lexical error
  - whle(f());
- It can significantly influence syntax analysis

Input buffering

- Lexical analysis takes 60-80% from the compile time
- One possible speedup: read the input file in blocks (buffers), the automaton works in the buffer memory
- Problems
  - Including a file means „including“ a buffer
    - #include

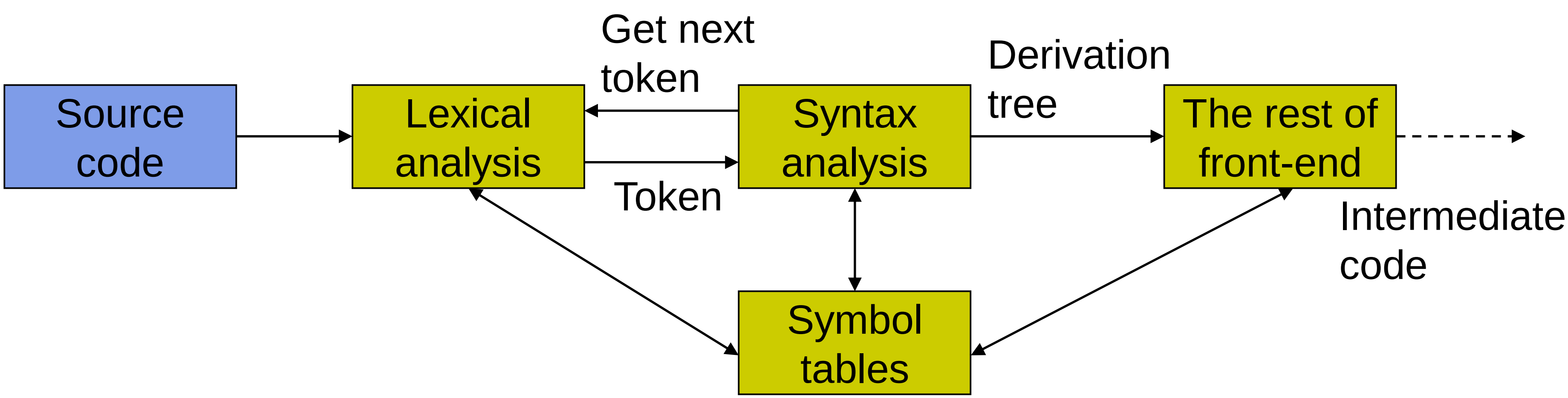
Compiler principles

Syntax analysis

Jakub Yaghob

Syntax analysis

- The main task
  - Decide, whether an input word is a word from an input language
- Other important tasks
  - Syntax-directed translation is the main loop of the compiler
  - Build the derivation tree
- Automaton type
  - We are talking about (deterministic) context-free grammars, therefore we are using (deterministic) pushdown automata



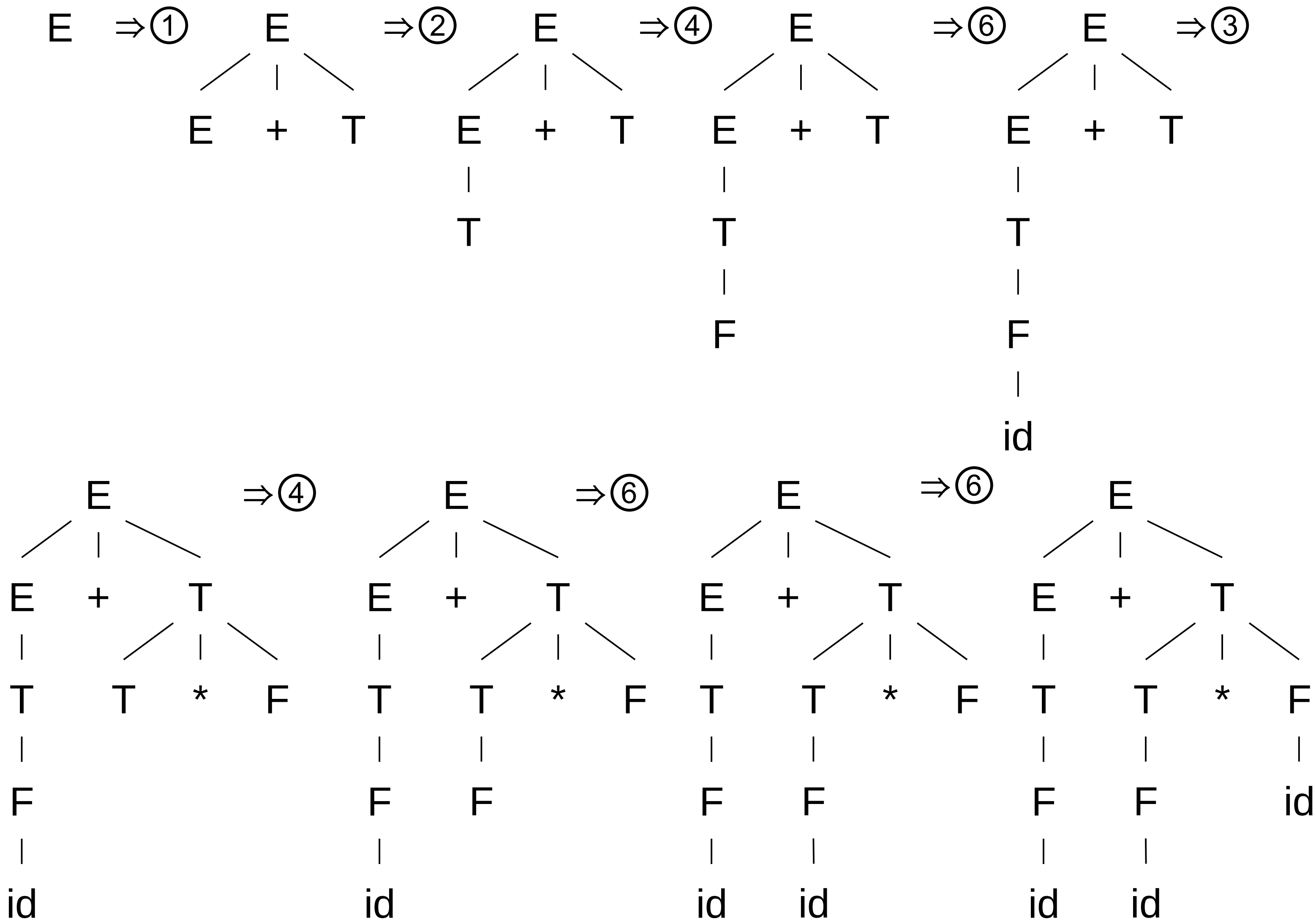
Our grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \mathbf{id}$

Derivation (parse, syntax) tree

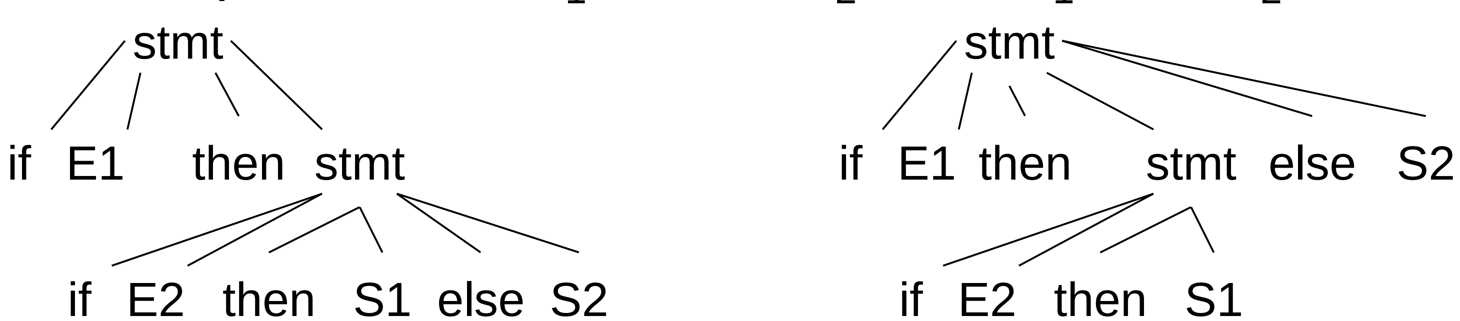
- Graphical representation of derivations using trees
  - Vertices are both non-terminals and terminals
  - Edges from inner vertex representing a non-terminal on the left side of a production rule to all symbols from the right side of a production rule
- E ⇒① E+T ⇒② T+T ⇒④ F+T ⇒⑥ id+T ⇒③ id+T*F ⇒④ id+F*F ⇒⑥ id+id*F ⇒⑥ id+id*id

Example



Ambiguous grammar

- We can construct distinct derivation trees for the same input word
- Real-life example (dangling else):
  - stmt → **if** expr **then** stmt
    | **if** expr **then** stmt **else** stmt
    | **while** expr **do** stmt
    | **goto** num
  - Input word: **if** E_1 **then if** E_2 **then** S_1 **else** S_2



Disambiguation

- Clarify, which tree is the right one
- In our case: **else** pairs with nearest “free” **if** (without **else**)
- Idea: “paired” statement is always between **if** and **else**
  - stmt → m_stmt
  - | u_stmt
  - m_stmt → **if** expr **then** m_stmt **else** m_stmt
    | **while** expr **do** m_stmt
    | **goto** num
  - u_stmt → **if** expr **then** stmt
    | **if** expr **then** m_stmt **else** u_stmt
    | **while** expr **do** u_stmt

Left recursion elimination

- A grammar is a left-recursive grammar, when there is a non-terminal A for which it is true that $A \Rightarrow^{+} A\alpha$ for a string α
- It is a problem for top-down parsing
- A simple solution for $\beta\alpha^{m}$:
  - $A \rightarrow A\alpha$
  - $A \rightarrow \beta$
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A'$
  - $A' \rightarrow \Lambda$

Removing left recursion from our grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

1. $E \rightarrow TE'$
2. $E' \rightarrow + TE'$
3. $E' \rightarrow \Lambda$
4. $T \rightarrow FT'$
5. $T' \rightarrow * FT'$
6. $T' \rightarrow \Lambda$
7. $F \rightarrow (E)$
8. $F \rightarrow id$

Left factoring

- It is not clear, which option we should choose

  - $A \rightarrow \alpha\beta_1$
  - $A \rightarrow \alpha\beta_2$

  - $A \rightarrow \alpha A'$
  - $A' \rightarrow \beta_1$
  - $A' \rightarrow \beta_2$

Non-context-free language constructions

- L_1={ wcw | w=(a|b)* }
  - Check, whether an identifier **w** is declared before using
- L_2={ $a^n b^m c^n d^m$ | n≥1, m≥1 }
  - Check, whether number of parameters in function call confirms to the function declaration
- L_3={ $a^n b^n c^n$ | n≥0 }
  - The problem of "underscoring" a word
    - **a** is a char, **b** is BS, **c** is underscore
  - (abc)* is a regular expression

Operators FIRST and FOLLOW – definitions

- If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If α can be derived to Λ, then Λ is also in FIRST(α)
- Define FOLLOW(A), for nonterminal A, to be the set of terminals that can appear immediately to the right of A in some string, where exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β. If A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A).

Construction of the FIRST operator

- Construction for a grammar symbol X
  - If X is terminal, then FIRST(X)={X}
  - If $X \rightarrow \Lambda$ is a production, then add Λ to FIRST(X)
  - If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place **a** in FIRST(X), if for some i, **a** is in FIRST(Y_i) and $\Lambda \in$ FIRST(Y_j) $\forall$ j<i. If $\Lambda \in$ FIRST(Y_j) $\forall$ j, then add Λ to FIRST(X)
- Construction for any string
  - The construction of the FIRST operator for a string $X_1 X_2 \ldots X_n$ is similar as for nonterminal.

Construction of the FOLLOW operator

- Construction for a nonterminal A
  - Place \$ in FOLLOW(S), where S is the start symbol of a grammar and \$ is EOS
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for Λ is placed in FOLLOW(B)
  - If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where $\Lambda \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW – an example for our grammar

- FIRST(E)={ (, id }
- FIRST(T)={ (, id }
- FIRST(F)={ (, id }
- FIRST(E')={ +, Λ }
- FIRST(T')={ *, Λ }

- FOLLOW(E)={), $ }
- FOLLOW(E')={), $ }
- FOLLOW(T)={ +,), $ }
- FOLLOW(T')={ +,), $ }
- FOLLOW(F)={ +, *,), $ }

Top-down parsing

- An attempt to find a leftmost derivation for an input string
- An attempt to construct a parse tree for the input starting from the root and creating the nodes of the tree in preorder
- Recursive-descent parsing
  - Recursive descent using procedures
- Nonrecursive predictive parsing
  - An automaton with an explicit stack
- Both solutions have a problem with left recursion in a grammar
- Many current parser generators use top-down parsing
  - ANTLR, CocoR – LL(1) grammars with conflict resolution using dynamic look-ahead expansion to LL(k)

Recursive-descent parsing

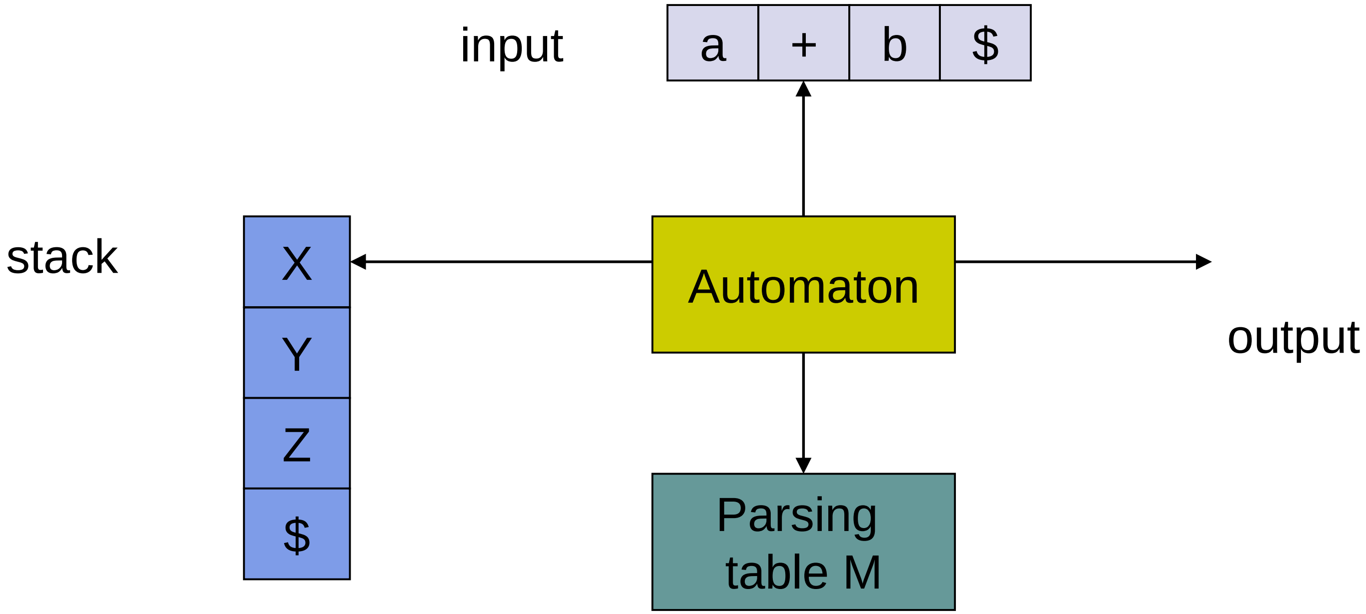
- One procedure/function for each nonterminal of a grammar
- Each procedure does two things
  - It decides, which grammar production with given nonterminal on the left side will be used using look-ahead. A production with right side α will be used, when the look-ahead is in FIRST(α). If there is a conflict for the look-ahead among some production right sides, the grammar is not suitable for recursive-descent parsing. A production with Λ on the right side will be used, if the look-ahead is not in FIRST of any right side.
  - Procedure code copies the right side of a production. Nonterminal means calling a procedure for this nonterminal. Terminal is compared with the look-ahead. If they are equal, a next terminal is read. If they are not equal, it is an error.

Recursive-descent parsing – example for our grammar

```
void match(token t) {
  if(lookahead==t)
    lookahead = nexttoken();
  else error();
}
void E(void) {
  T(); Eap();
}
void Eap(void) {
  if(lookahead=='+') {
    match('+'); T(); Eap();
  }
}
void T(void) {
  F(); Tap();
}
```

```
void Tap(void) {
  if(lookahead=='*') {
    match('*'); F(); Tap(); }
}
void F(void) {
  switch(lookahead) {
    case '(':match('('); E();
             match(')');break;
    case 'id':
       match('id'); break;
    default:
       error();
  }
}
```

Nonrecursive predictive parsing



- Parsing table M[A, a], where A is nonterminal and **a** is terminal
- The stack contains grammar symbols

LL(1) automaton behavior

- Initial configuration
  - Input pointer points to the first terminal in the input string
  - The stack contains the start symbol of the grammar on top of $
- In each step, the automaton decides, what to do, using a symbol X on top of the stack and a terminal **a**, pointed by input pointer
  - If $X=a=\$$, the parser halts, parsing finished successfully
  - If $X=a\neq\$$, the parser pops X from the stack and advances the input pointer to the next input symbol
  - If $X\neq a$ and $X\in T$, the parser reports error
  - If X is a nonterminal, the parser uses entry M[X, a]. If this entry is a production, the parser replaces X on top of the stack by the right side (leftmost symbol on top of the stack). At the same time, the parser generates an output about using the production. If the entry is **error**, the parser informs about a syntax error.

Construction of predictive parsing tables

- For each production $A \to \alpha$ do following steps
  - For $\forall$ $a \in FIRST(\alpha)$ add $A \to \alpha$ to $M[A, a]$
  - If $\Lambda \in FIRST(\alpha)$, add $A \to \alpha$ to $M [A, b]$ $\forall$ $b \in FOLLOW(A)$. Moreover, if $\$ \in FOLLOW(A)$, add $A \to \alpha$ to $M[A, \$]$
- Mark each empty entry in M as **error**

Example of table construction for our grammar

	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→Λ	E’→Λ
T	T→FT’			T→FT’		
T’		T’→Λ	T’→*FT’		T’→Λ	T’→Λ
F	F→id			F→(E)		

Example of parser behavior for our grammar

Stack	Input	Output
$E	id+id*id$	
$E’T	id+id*id$	E→TE’
$E’T’F	id+id*id$	T→FT’
$E’T’id	id+id*id$	F→id
$E’T’	+id*id$	
$E’	+id*id$	T’→Λ
$E’T+	+id*id$	E’→+TE’
$E’T	id*id$	
$E’T’F	id*id$	T→FT’

Stack	Input	Output
$E’T’id	id*id$	F→id
$E’T’	*id$	
$E’T’F*	*id$	T’→*FT’
$E’T’F	id$	
$E’T’id	id$	F→id
$E’T’	$	
$E’	$	T’→Λ
$	$	E’→Λ

LL(1) grammar

- Context-free grammar $G=(T,N,S,P)$ is a LL(1) grammar, if and only if whenever $A \rightarrow \alpha$, $A \rightarrow \beta \in P$ are two distinct ($\alpha \neq \beta$) productions of G and we have any left sentential forms $uA\gamma$, $vA\delta$, where $u,v \in T^*$ and $\gamma,\delta \in (T \cup N)^*$, the following condition holds:
  - $FIRST(\alpha\gamma) \cap FIRST(\beta\delta) = \emptyset$
- Simplified detection: no ambiguous or left-recursive grammar can be LL(1)

Grammar terminology

- PXY(k)
- X – direction of the input reading
  - In our case always L, i.e. from left to right
- Y – kind of derivation
  - L – left derivation
  - R – right derivation
- P – prefix
  - Subtle division of some grammar classes
- k – look-ahead
  - An integer, usually 1, can be 0 or more generally k
- Examples
  - LL(1), LR(0), LR(1), LL(k), SLR(1), LALR(1)

Expanding definition of FIRST and FOLLOW on k

- If α is a string from grammar symbols, then $FIRST_k(\alpha)$ is a set of terminal words with maximal length k, which are on the beginning of at least one string derived from α. If α can be derived on Λ, then Λ is in $FIRST_k(\alpha)$.
- $FOLLOW_k(A)$ for nonterminal A is a set of terminal words with maximal length k, which can be on the right side of A in any string derived from the start nonterminal ($S \Rightarrow^* \alpha A u \beta$ for some α and β). If A is the right-most symbol in any sentential form, then $ is in $FOLLOW_k(A)$.

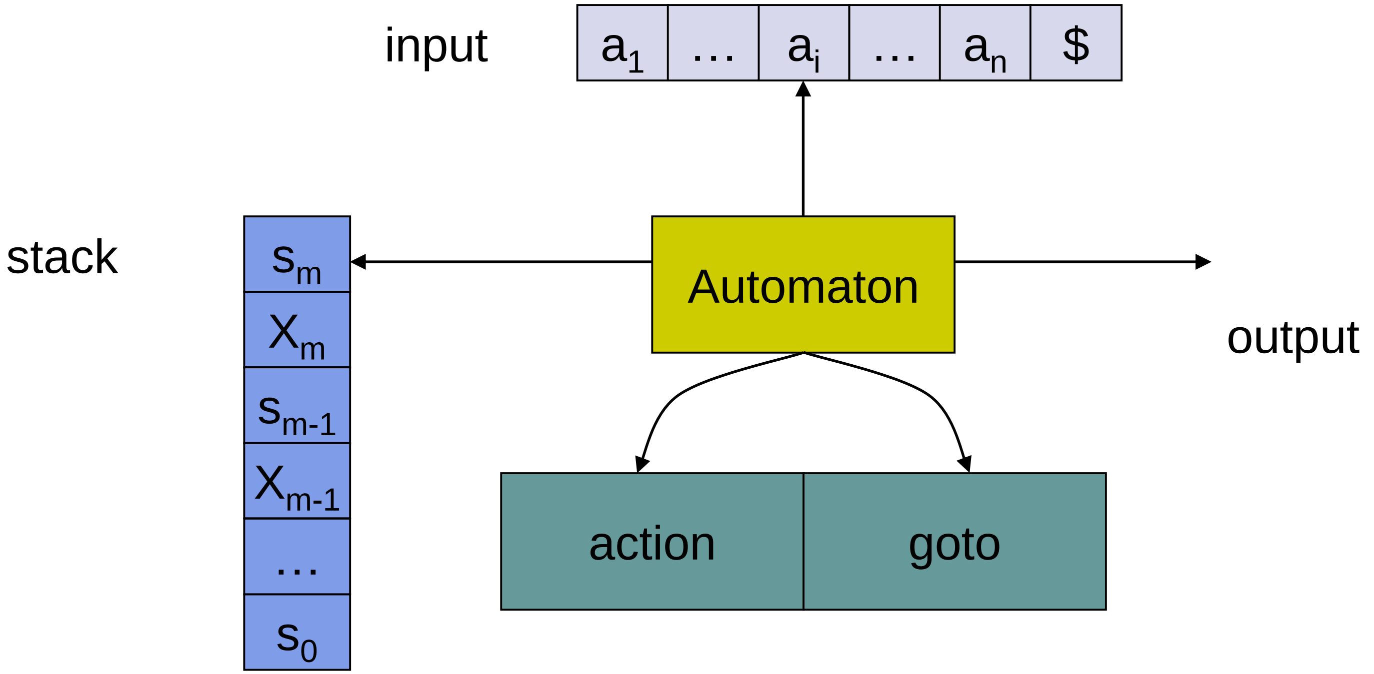
LL(k) grammar

- Context-free grammar $G=(T,N,S,P)$ is a strong LL(k) grammar for $k \geq 1$, if and only if whenever $A \rightarrow \alpha$, $A \rightarrow \beta \in P$ are two distinct ($\alpha \neq \beta$) productions and we have any left sentential forms $uA\gamma$, $vA\delta$, where $u,v \in T^*$ and $\gamma,\delta \in (T \cup N)^*$, the following condition holds:
  - $FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\delta) = \emptyset$.
- LL(k) (not strong)
  - $u=v$, $\gamma=\delta$

Bottom-up analysis

- Attempts to find in reverse the rightmost derivation for an input string
- Attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root
- Replace a substring corresponding to a right side of a production by a nonterminal from the left side of the production in each reduce step
- Used in parser generators
  - Bison – LALR(1), GLR(1)
- Advantages against LL(1) parsers
  - It can be implemented with the same efficiency as top-down parsing
  - The class of decidable languages LR(1) is a proper superset of LL(1)
- SLR(1), LR(1), LALR(1)

LR parser automaton



- s_i are states
  - A state on the top of the stack is the current state of the automaton
- x_i are grammar symbols

LR(1) automaton behavior

- Initial configuration
  - Input pointer points to the first terminal in the input string
  - Initial state s_0 is on the stack
- In each step address table action[s_m, a_i] using s_m and a_i
  - Shift s, where s is a new state
    - It shifts the input tape by 1 terminal and add a_i and s on the top of the stack
  - Reduce using production $A \rightarrow \alpha$
    - Remove r=|α| pairs (s_k, X_k) from the top of the stack, add A on the top of the stack and then goto[s_{m-r}, A] (s_{m-r} is a state on the top of the stack after erasing pairs)
    - Generate an output
  - Accept
    - The input string is accepted
    - Generate an output
  - Error
    - The input string is not in the input language

LR automaton tables for our grammar

state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example of LR parser behavior

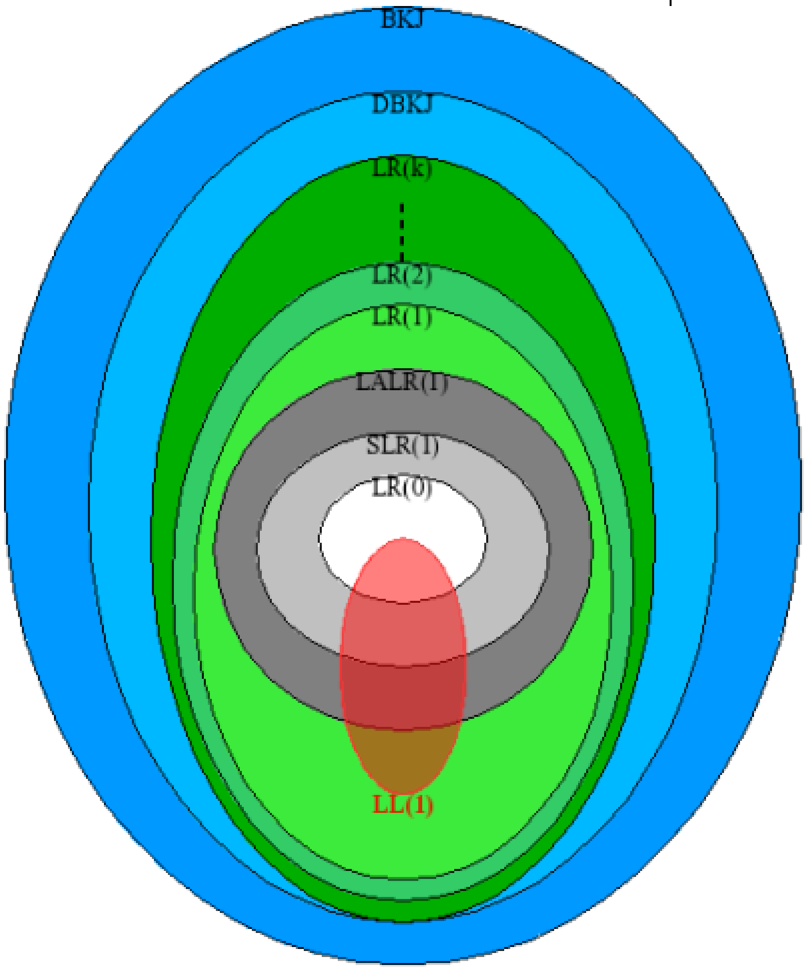
Stack	Input	Action
0	**id+id*id$**	s5
0 **id** 5	**+id*id$**	r6: F→**id**
0 F 3	**+id*id$**	r4: T→F
0 T 2	**+id*id$**	r2: E→T
0 E 1	**+id*id$**	s6
0 E 1 **+** 6	**id*id$**	s5
0 E 1 **+** 6 **id** 5	***id$**	r6: F→**id**
0 E 1 **+** 6 F 3	***id$**	r4: T→F
0 E 1 **+** 6 T 9	***id$**	s7
0 E 1 **+** 6 T 9 ***** 7	**id$**	s5
0 E 1 **+** 6 T 9 ***** 7 **id** 5	**$**	r6: F→**id**
0 E 1 **+** 6 T 9 ***** 7 F 10	**$**	r3: T→T ***** F
0 E 1 **+** 6 T 9	**$**	r1: E→E **+** T
0 E 1	**$**	acc

LR(k) grammar

- Context-free grammar $G=(T,N,S,P)$ is LR(k) grammar for $k \geq 1$, if and only if whenever $A \longrightarrow \alpha$, $A \longrightarrow \beta \in P$ are two distinct ($\alpha \neq \beta$) productions of G and we have any two right sentential forms $\gamma A u$, $\delta A v$, where $u,v \in T^*$ and $\gamma,\delta \in (T \cup N)^*$, the following condition holds:
  - $FIRST_k(u) \cap FIRST_k(v) = \varnothing$

Grammars (languages) strength

- Union of all LR(k) are deterministic context-free languages (DBKJ) and it is a proper subset of all context-free languages (BKJ)



Grammar augmentation

- Augmentation of a grammar $G=(T,N,S,P)$ is a grammar $G'=(T,N',S',P')$, where $N'=N\cup\{S'\}$ and $P'=P\cup\{S'\rightarrow S\}$
- The augmentation is not necessary whenever S is on the left side of one production and it isn't on any right side of grammar productions
- It helps recognize the end of parsing
- For our grammar:
  - $S'\rightarrow E$

LR(0) items

- LR(0) item of a grammar G is a production with a special symbol dot on the right side
  - Special symbol is a valid symbol for comparison of two LR(0) items of a same production. LR(0) items of the same production are different, whenever the dot is on different position. Moreover, the dot is not a grammar symbol
- An example for production E → E + T:

E → ♦E + T

E → E ♦+ T

E → E + ♦T

E → E + T♦

The closure operation

- If I is a set of LR(0) items for a grammar G, then CLOSURE(I) is a set of LR(0) items constructed from I by following rules:
  - Add I to the CLOSURE(I)
  - $\forall$ $A \rightarrow \alpha \blacklozenge B\beta \in$ CLOSURE(I), where $B \in N$, add $\forall$ $B \rightarrow \gamma \in P$ to CLOSURE(I) LR(0) item $B \rightarrow \blacklozenge \gamma$, if it is not already there. Apply this rule until no more new LR(0) items can be added to CLOSURE(I)

Example of closure for our grammar

- I={S’→♦E}
- CLOSURE(I)=
  - S’→ ♦E
  - E → ♦E **+** T
  - E → ♦T
  - T → ♦T ***** F
  - T → ♦F
  - F → ♦**(** E **)**
  - F → ♦**id**

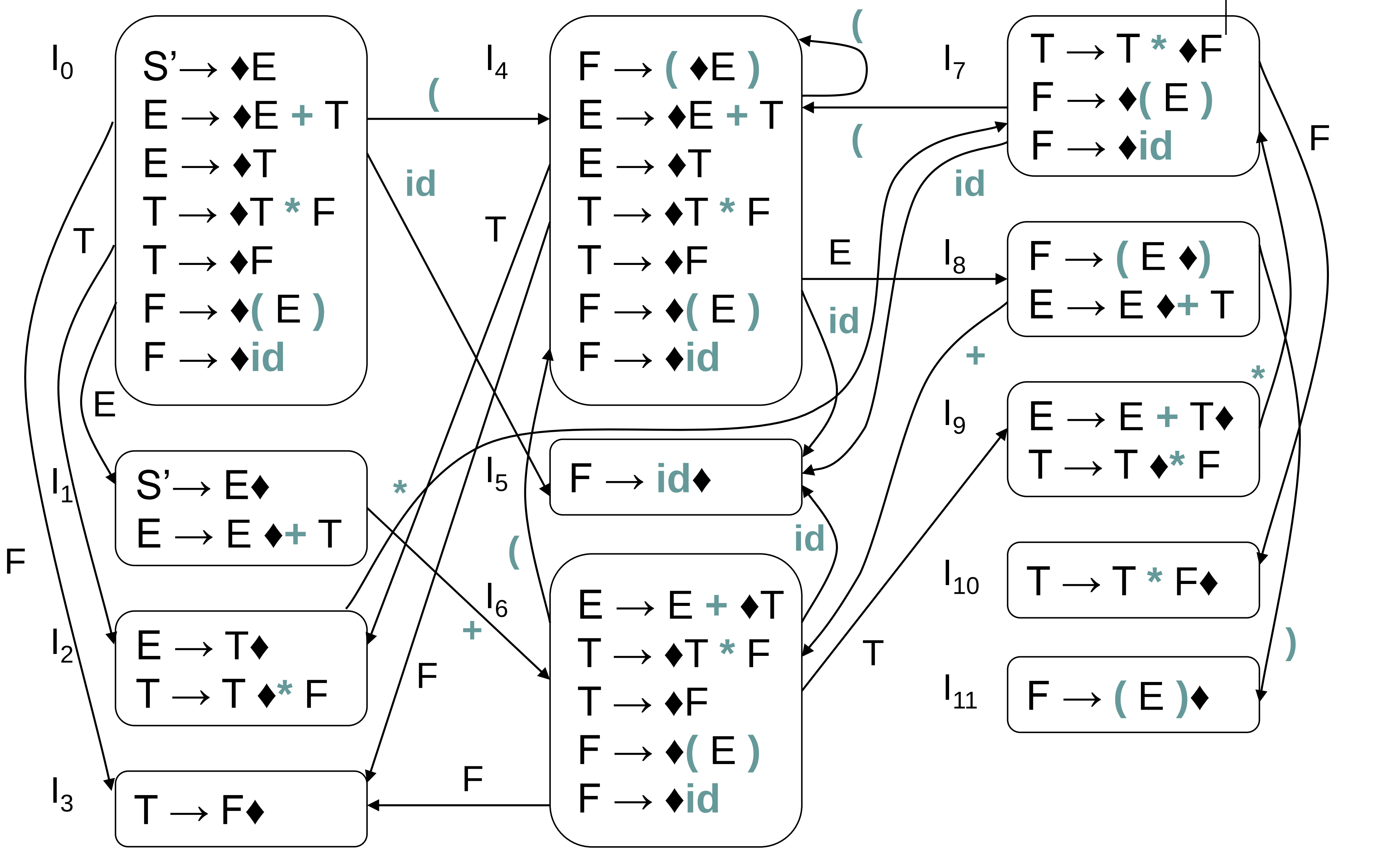
GOTO operation

- GOTO(I, X) operation for a set I of LR(0) items and a grammar symbol X is defined to be the closure of the set of all LR(0) items $A \to \alpha X \blacklozenge \beta$ such that $A \to \alpha \blacklozenge X \beta \in I$

Construction of canonical collection of sets of LR(0) items

- We have an augmented grammar $G'=(T,N',S',P')$
- Construction of canonical collection C of sets of LR(0) items:
  - We start with $C=\{ CLOSURE(\{S'\rightarrow\blacklozenge S\}) \}$
  - $\forall\ I\in C$ and $\forall\ X\in T\cup N'$ such as $GOTO(I, X)\notin C \wedge GOTO(I, X)\neq\emptyset$, add $GOTO(I, X)$ to C. Repeat this step, until something new is added to C.

Construction of canonical collection for our grammar



Valid items

- LR(0) item $A \rightarrow \beta_1 \blacklozenge \beta_2$ is a valid item for a viable prefix $\alpha\beta_1$, if there is a rightmost derivation $S' \Rightarrow^{+} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$
- It is a great hint for a parser. It helps to decide, if the parser should make a shift or a reduction, if $\alpha\beta_1$ is on top of the stack
- Basic LR parsing theorem: A set of valid items for a viable prefix γ is exactly a set of items reachable from the initial state through the prefix γ by deterministic finite automaton constructed from canonical collection with GOTO transitions.

SLR(1) automaton construction

- We have an augmented grammar G'. Tables of SLR(1) automaton are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(0) items
  - State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows
    - $A \rightarrow \alpha \blacklozenge a\beta \in I_i$, $a \in T \wedge GOTO(I_i,a)=I_j$, then action[i,a]=shift j
    - $A \rightarrow \alpha \blacklozenge \in I_i$, then $\forall a \in FOLLOW(A) \wedge A \neq S'$ is action[i,a]=reduce $A \rightarrow \alpha$
    - $S' \rightarrow S \blacklozenge \in I_i$, then action[i,$]=accept
  - If there is a conflict in the previous step, the grammar is not a SLR(1) grammar and the automaton cannot be constructed
  - Table goto is indexed by state *i* and $A \in N'$: whenever $GOTO(I_i,A)=I_j$, then goto [i,A]=j
  - All empty cells are filled by error instruction
  - The initial state of the parser is the state, which contains LR(0) item $S' \rightarrow \blacklozenge S$

Full LR(1) automata

- action[i,a] is set to reduction $A \rightarrow \alpha$, when $A \rightarrow \alpha \blacklozenge \in I_i$, $\forall a \in \text{FOLLOW}(A)$ for a state *i* during SLR(1) construction
- In some situation, when *i* is on top of the stack, the viable prefix $\beta\alpha$ is in form, where βA cannot be followed by a terminal **a** in any right sentential form. Therefore reduction $A \rightarrow \alpha$ is for lookahed **a** invalid.
- Solution: add more information to states, so we can avoid invalid reductions.

LR(1) items

- The added information is stored as an additional terminal for each LR(0) item. Such item has a form $[A \rightarrow \alpha \blacklozenge \beta, a]$, where $A \rightarrow \alpha\beta \in P$, $a \in T$, and we call it LR(1) item. The terminal **a** is called lookahead.
  - The lookahed has no meaning for $A \rightarrow \alpha \blacklozenge \beta$, where $\beta \neq \Lambda$
  - Reduction $A \rightarrow \alpha$ is set only when $[A \rightarrow \alpha \blacklozenge, a] \in I_i$ for current state *i* and a terminal **a** on the input
  - A set of terminals created from lokaheads of LR(1) items $\subseteq$FOLLOW(A)
- LR(1) item $[A \rightarrow \alpha \blacklozenge \beta, a]$ is valid for viable prefix γ, whenever $\exists$ right derivation $S \Rightarrow^{+} \delta A w \Rightarrow \delta \alpha \beta w$, where
  - $\gamma = \delta\alpha$
  - Either **a** is the first symbol of w or $w = \Lambda$ and **a** is \$

Closure for LR(1) items

- We have a set of LR(1) items I for a grammar G. We define CLOSURE1(I) as a set of LR(1) items constructed from I by following procedure:
  - Add set I to CLOSURE1(I)
  - $\forall$ $[A \rightarrow \alpha \blacklozenge B\beta, a] \in$ CLOSURE1(I), where $B \in N$, add LR(1) item $[B \rightarrow \blacklozenge \gamma, b]$ $\forall$ $B \rightarrow \gamma \in P$ and $\forall b \in$ FIRST(βa) to CLOSURE1(I), if it isn’t there already. Repeat this step, until something is added to CLOSURE1(I).

GOTO operation for LR(1) items

- We define GOTO1(I, X) operation for a set I of LR(1) items and a grammar symbol X as a CLOSURE1 of a set of all items $[A \rightarrow \alpha X \blacklozenge \beta, a]$ where $[A \rightarrow \alpha \blacklozenge X \beta, a] \in I$

Construction of canonical collection of sets of LR(1) items

- We have an augmented grammar G’=(T,N’,S’,P’)
- Construction of canonical collection C of LR(1) items:
  - We start with C={ CLOSURE1({[S’→♦S,$]}) }
  - Add GOTO1(I, X) to C $\forall$ $I \in C$ and $\forall$ $X \in T \cup N'$, where $GOTO1(I, X) \notin C \wedge GOTO1(I, X) \neq \emptyset$. Repeat this step, until something new is added to C.

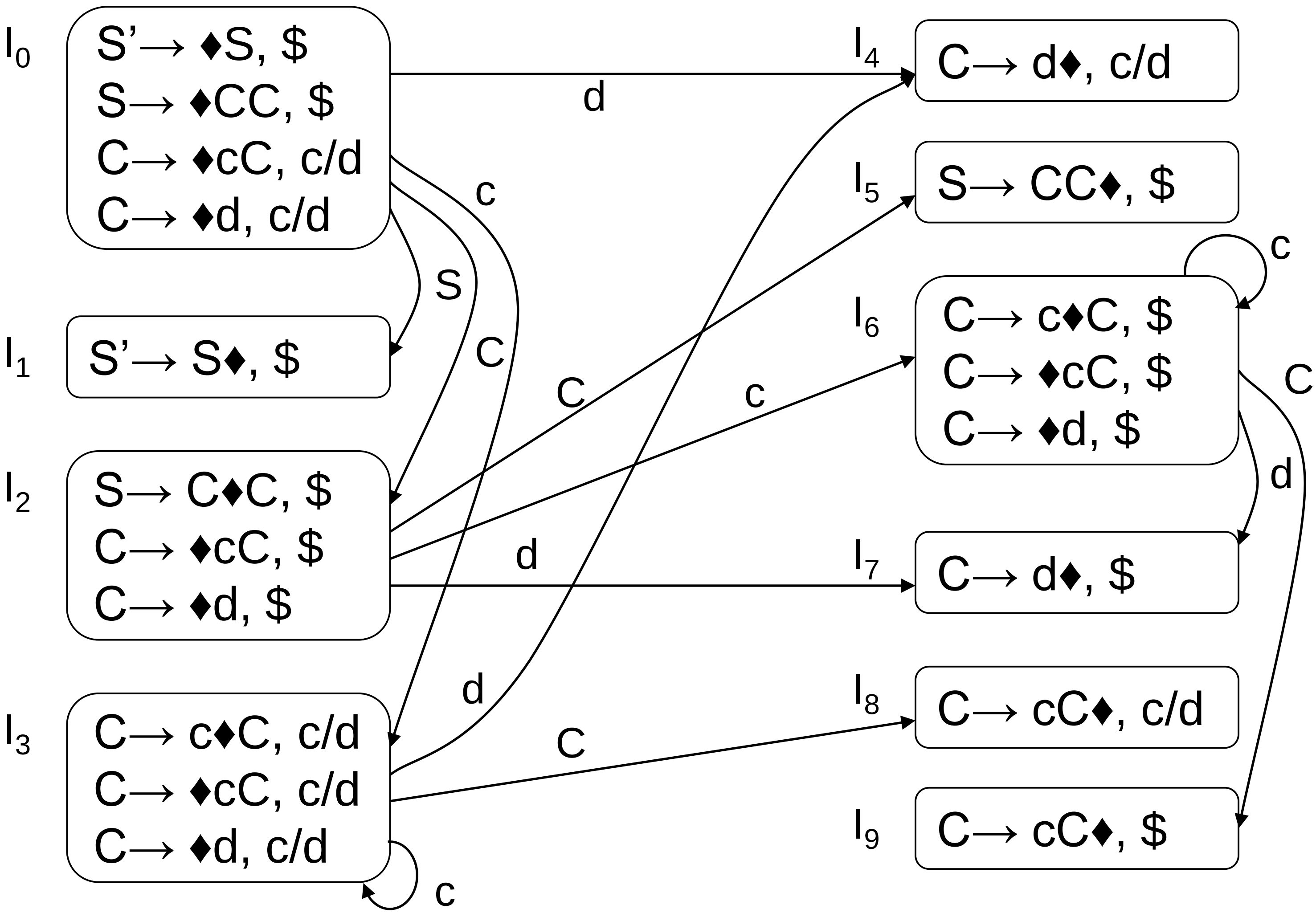
Example of LR(1) grammar, which is not SLR(1)

- $S' \rightarrow S$
- $S \rightarrow CC$
- $C \rightarrow cC$
- $C \rightarrow d$

Example of closure construction for LR(1) items

- I={[S’→♦S,$]}
- CLOSURE1(I)=
  - S’→ ♦S, $ β=Λ,FIRST(β$)=FIRST($)={$}
  - S→ ♦CC, $ β=C,FIRST(C$)={c,d}
  - C→ ♦cC, c/d
  - C→ ♦d, c/d

Example of construction of canonical collection of LR(1) items



LR(1) parser construction

- We have an augmented grammar G'. LR(1) automaton tables are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(1) items
  - State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows
    - $[A \to \alpha \blacklozenge a\beta, b] \in I_i, a \in T \wedge \text{GOTO1}(I_i, a) = I_j$, then action[i,a]=shift j
    - $[A \to \alpha \blacklozenge, a] \in I_i \wedge A \neq S'$, then action[i,a]=reduce $A \to \alpha$
    - $[S' \to S \blacklozenge, \$] \in I_i$, then action[i,\$]=accept
  - If there is a conflict in the previous step, the grammar is not a LR(1) grammar and the automaton cannot be constructed
  - Table goto is indexed by state *i* and $A \in N'$: whenever $\text{GOTO1}(I_i, A) = I_j$, then goto [i,A]=j
  - All empty cells are filled by error instruction
  - The initial state of the parser is the state, which contains LR(1) item $[S' \to \blacklozenge S, \$]$

LALR

- LALR=LookAhead-LR
- Often used in practice
  - Bison
  - Most common programming languages can be expressed by an LALR grammar
  - Parser tables are considerably smaller then LR(1) tables
- SLR and LALR parsers have the same number of states, LR parsers have greater number of states
  - Common languages have hundreds of states
  - LR(1) parsers have thousands of states for the same grammar

How to make smaller tables?

- Idea: merge sets with the same core into one set including GOTO1 merge
  - Core: a set of LR(0) items (no lookahead)
  - Merge cannot produce shift/reduce conflict
    - Suppose in the union there is a conflict on lookahead **a** for LR(1) items $[A \to \alpha \blacklozenge, a]$ and $[B \to \beta \blacklozenge a\gamma, b]$
    - Cores are same, therefore in the set with $[A \to \alpha \blacklozenge, a]$ must be $[B \to \beta \blacklozenge a\gamma, c]$ as well for some **c**. There was already a shift/reduce conflict before merge
  - Merge can produce reduce/reduce conflict

Easy LALR(1) table construction

- We have an augmented grammar G'. LALR(1) automaton tables are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(1) items
  - For each core in collection C, find all sets having that core, and replace these sets by their union
  - Let C'={ J_0, J_1, …, J_m } be the resulting collection of LR(1) items
  - Table action is constructed for C' in the same manner as for full LR(1) parser
  - If there is a conflict, the grammar is not LALR(1) grammar
  - If J∈C' is the union of sets of LR(1) items I_i ($J=I_1 \cup I_2 \cup \ldots I_k$), then cores GOTO1($I_1$,X), …, GOTO1($I_k$,X) are the same, since I_1, …, I_k all have the same core. Let K be the union of all sets of items having the same core as goto(I_1,X). Then GOTO1(J,X)=K
- Important disadvantage – we need to construct full LR(1)

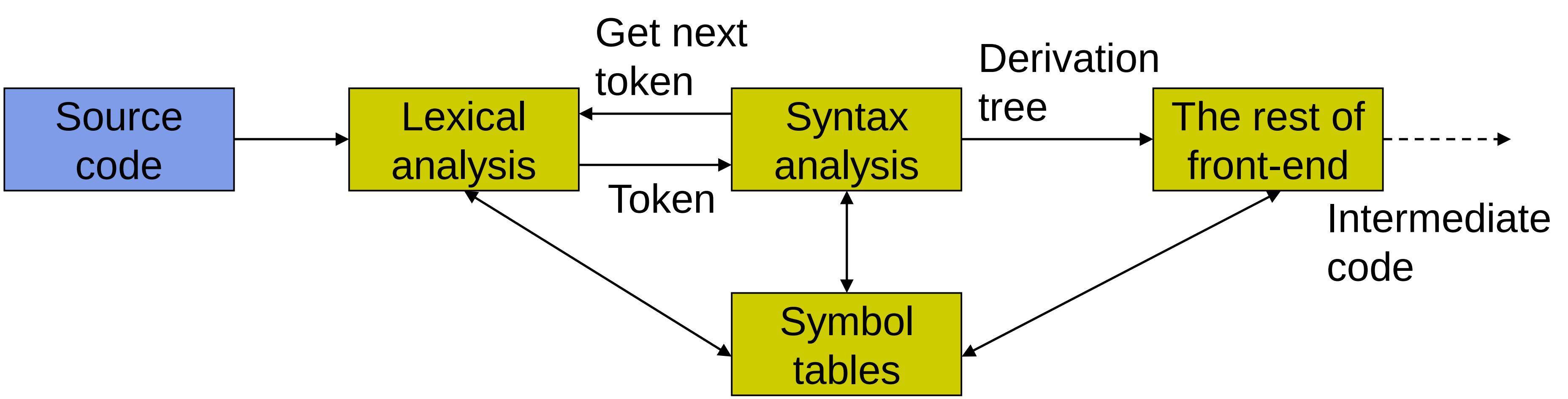
Compiler principles

Syntax analysis

Jakub Yaghob

Syntax analysis

- The main task
  - Decide, whether an input word is a word from an input language
- Other important tasks
  - Syntax-directed translation is the main loop of the compiler
  - Build the derivation tree
- Automaton type
  - We are talking about (deterministic) context-free grammars, therefore we are using (deterministic) pushdown automata



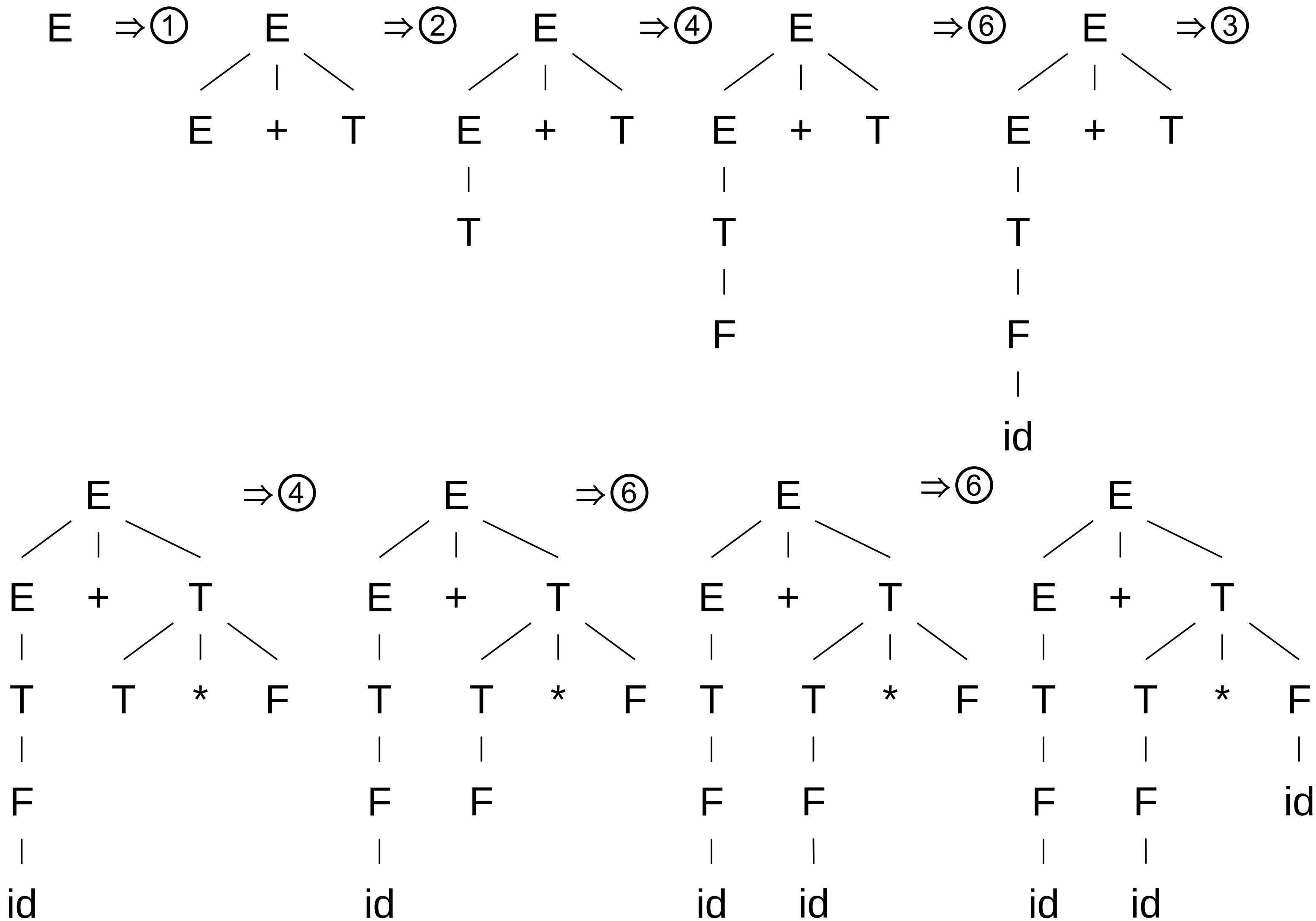
Our grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \mathbf{id}$

Derivation (parse, syntax) tree

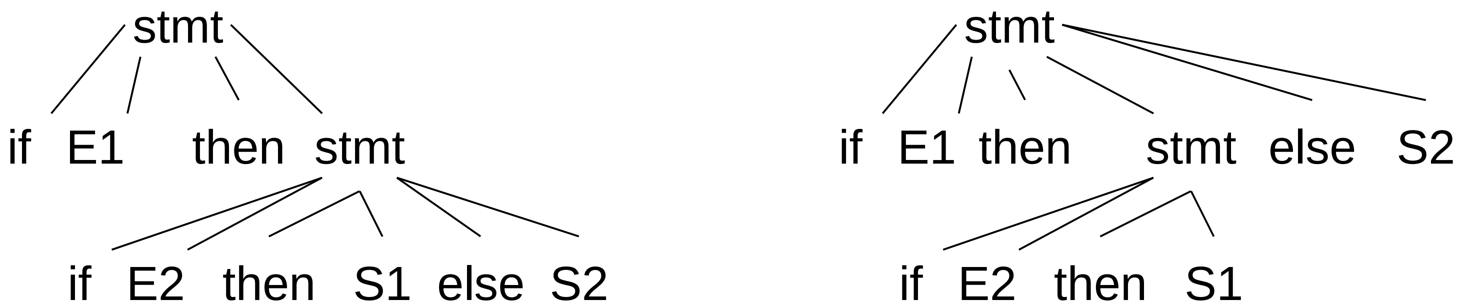
- Graphical representation of derivations using trees
  - Vertices are both non-terminals and terminals
  - Edges from inner vertex representing a non-terminal on the left side of a production rule to all symbols from the right side of a production rule
- E $\Rightarrow$① E+T $\Rightarrow$② T+T $\Rightarrow$④ F+T $\Rightarrow$⑥ id+T $\Rightarrow$③ id+T*F $\Rightarrow$④ id+F*F $\Rightarrow$⑥ id+id*F $\Rightarrow$⑥ id+id*id

Example



Ambiguous grammar

- We can construct distinct derivation trees for the same input word
- Real-life example (dangling else):
  - stmt → **if** expr **then** stmt
    | **if** expr **then** stmt **else** stmt
    | **while** expr **do** stmt
    | **goto** num
  - Input word: **if** E_1 **then if** E_2 **then** S_1 **else** S_2



Disambiguation

- Clarify, which tree is the right one
- In our case: **else** pairs with nearest “free” **if** (without **else**)
- Idea: “paired” statement is always between **if** and **else**
  - stmt → m_stmt
  - | u_stmt
  - m_stmt → **if** expr **then** m_stmt **else** m_stmt
    | **while** expr **do** m_stmt
    | **goto** num
  - u_stmt → **if** expr **then** stmt
    | **if** expr **then** m_stmt **else** u_stmt
    | **while** expr **do** u_stmt

Left recursion elimination

- A grammar is a left-recursive grammar, when there is a non-terminal A for which it is true that $A \Rightarrow^{+} A\alpha$ for a string α
- It is a problem for top-down parsing
- A simple solution for $\beta\alpha^{m}$:
  - $A \rightarrow A\alpha$
  - $A \rightarrow \beta$

  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A'$
  - $A' \rightarrow \Lambda$

Removing left recursion from our grammar

1. E $\rightarrow$ E **+** T
2. E $\rightarrow$ T
3. T $\rightarrow$ T ***** F
4. T $\rightarrow$ F
5. F $\rightarrow$ **(** E **)**
6. F $\rightarrow$ **id**

1. E $\rightarrow$ TE’
2. E’ $\rightarrow$ **+** TE’
3. E’ $\rightarrow$ Λ
4. T $\rightarrow$ FT’
5. T’ $\rightarrow$ ***** FT’
6. T’ $\rightarrow$ Λ
7. F $\rightarrow$ **(** E **)**
8. F $\rightarrow$ **id**

Left factoring

- It is not clear, which option we should choose

  - $A \rightarrow \alpha\beta_1$
  - $A \rightarrow \alpha\beta_2$

  - $A \rightarrow \alpha A'$
  - $A' \rightarrow \beta_1$
  - $A' \rightarrow \beta_2$

Non-context-free language constructions

- $L_1=\{ wcw \mid w=(a|b)^* \}$
  - Check, whether an identifier **w** is declared before using
- $L_2=\{ a^n b^m c^n d^m \mid n \geq 1, m \geq 1 \}$
  - Check, whether number of parameters in function call confirms to the function declaration
- $L_3=\{ a^n b^n c^n \mid n \geq 0 \}$
  - The problem of “underscoring” a word
    - **a** is a char, **b** is BS, **c** is underscore
  - (abc)* is a regular expression

Operators FIRST and FOLLOW – definitions

- If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If α can be derived to Λ, then Λ is also in FIRST(α)
- Define FOLLOW(A), for nonterminal A, to be the set of terminals that can appear immediately to the right of A in some string, where exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β. If A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A).

Construction of the FIRST operator

- Construction for a grammar symbol X
  - If X is terminal, then FIRST(X)={X}
  - If $X \rightarrow \Lambda$ is a production, then add Λ to FIRST(X)
  - If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place **a** in FIRST(X), if for some i, **a** is in FIRST(Y_i) and $\Lambda \in$FIRST(Y_j) $\forall$ j<i. If $\Lambda \in$FIRST(Y_j) $\forall$ j, then add Λ to FIRST(X)
- Construction for any string
  - The construction of the FIRST operator for a string $X_1 X_2 \ldots X_n$ is similar as for nonterminal.

Construction of the FOLLOW operator

- Construction for a nonterminal A
  - Place \$ in FOLLOW(S), where S is the start symbol of a grammar and \$ is EOS
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for Λ is placed in FOLLOW(B)
  - If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where $\Lambda \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW – an example for our grammar

- FIRST(E)={ (, id }
- FIRST(T)={ (, id }
- FIRST(F)={ (, id }
- FIRST(E')={ +, Λ }
- FIRST(T')={ *, Λ }

- FOLLOW(E)={), $ }
- FOLLOW(E')={), $ }
- FOLLOW(T)={ +,), $ }
- FOLLOW(T')={ +,), $ }
- FOLLOW(F)={ +, *,), $ }

Top-down parsing

- An attempt to find a leftmost derivation for an input string
- An attempt to construct a parse tree for the input starting from the root and creating the nodes of the tree in preorder
- Recursive-descent parsing
  - Recursive descent using procedures
- Nonrecursive predictive parsing
  - An automaton with an explicit stack
- Both solutions have a problem with left recursion in a grammar
- Many current parser generators use top-down parsing
  - ANTLR, CocoR – LL(1) grammars with conflict resolution using dynamic look-ahead expansion to LL(k)

Recursive-descent parsing

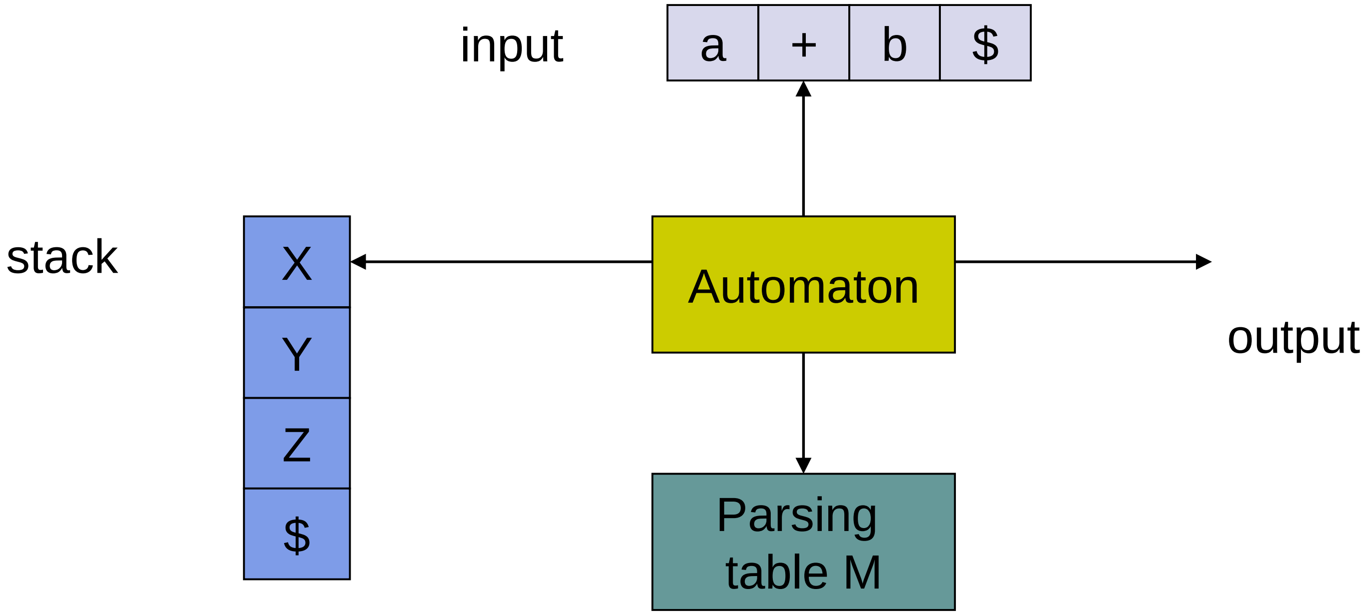
- One procedure/function for each nonterminal of a grammar
- Each procedure does two things
  - It decides, which grammar production with given nonterminal on the left side will be used using look-ahead. A production with right side α will be used, when the look-ahead is in FIRST(α). If there is a conflict for the look-ahead among some production right sides, the grammar is not suitable for recursive-descent parsing. A production with Λ on the right side will be used, if the look-ahead is not in FIRST of any right side.
  - Procedure code copies the right side of a production. Nonterminal means calling a procedure for this nonterminal. Terminal is compared with the look-ahead. If they are equal, a next terminal is read. If they are not equal, it is an error.

Recursive-descent parsing – example for our grammar

```
void match(token t) {
  if(lookahead==t)
    lookahead = nexttoken();
  else error();
}
void E(void) {
  T(); Eap();
}
void Eap(void) {
  if(lookahead=='+') {
    match('+'); T(); Eap();
  }
}
void T(void) {
  F(); Tap();
}
```

```
void Tap(void) {
  if(lookahead=='*') {
    match('*'); F(); Tap(); }
}
void F(void) {
  switch(lookahead) {
    case '(':match('('); E();
             match(')');break;
    case 'id':
       match('id'); break;
    default:
       error();
  }
}
```

Nonrecursive predictive parsing



- Parsing table M[A, a], where A is nonterminal and **a** is terminal
- The stack contains grammar symbols

LL(1) automaton behavior

- Initial configuration
  - Input pointer points to the first terminal in the input string
  - The stack contains the start symbol of the grammar on top of $
- In each step, the automaton decides, what to do, using a symbol X on top of the stack and a terminal **a**, pointed by input pointer
  - If $X=a=\$$, the parser halts, parsing finished successfully
  - If $X=a\neq\$$, the parser pops X from the stack and advances the input pointer to the next input symbol
  - If $X\neq a$ and $X\in T$, the parser reports error
  - If X is a nonterminal, the parser uses entry M[X, a]. If this entry is a production, the parser replaces X on top of the stack by the right side (leftmost symbol on top of the stack). At the same time, the parser generates an output about using the production. If the entry is **error**, the parser informs about a syntax error.

Construction of predictive parsing tables

- For each production $A \to \alpha$ do following steps
  - For $\forall\ a \in \text{FIRST}(\alpha)$ add $A \to \alpha$ to M[A, a]
  - If $\Lambda \in \text{FIRST}(\alpha)$, add $A \to \alpha$ to M [A, b] $\forall\ b \in \text{FOLLOW}(A)$. Moreover, if $\$ \in \text{FOLLOW}(A)$, add $A \to \alpha$ to M[A, $]
- Mark each empty entry in M as **error**

Example of table construction for our grammar

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \Lambda$	$E' \rightarrow \Lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \Lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \Lambda$	$T' \rightarrow \Lambda$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Example of parser behavior for our grammar

Stack	Input	Output
$E	**id+id*id$**	
$E'T	**id+id*id$**	E→TE'
$E'T'F	**id+id*id$**	T→FT'
$E'T'**id**	**id+id*id$**	F→**id**
$E'T'	**+id*id$**	
$E'	**+id*id$**	T'→Λ
$E'T**+**	**+id*id$**	E'→**+**TE'
$E'T	**id*id$**	
$E'T'F	**id*id$**	T→FT'

Stack	Input	Output
$E'T'**id**	**id*id$**	F→**id**
$E'T'	***id$**	
$E'T'F*****	***id$**	T'→*****FT'
$E'T'F	**id$**	
$E'T'**id**	**id$**	F→**id**
$E'T'	**$**	
$E'	**$**	T'→Λ
$	**$**	E'→Λ

LL(1) grammar

- Context-free grammar $G=(T,N,S,P)$ is a LL(1) grammar, if and only if whenever $A \rightarrow \alpha$, $A \rightarrow \beta \in P$ are two distinct ($\alpha \neq \beta$) productions of G and we have any left sentential forms $uA\gamma$, $vA\delta$, where $u,v \in T^*$ and $\gamma,\delta \in (T \cup N)^*$, the following condition holds:
  - $\mathrm{FIRST}(\alpha\gamma) \cap \mathrm{FIRST}(\beta\delta) = \emptyset$
- Simplified detection: no ambiguous or left-recursive grammar can be LL(1)

Grammar terminology

- PXY(k)
- X – direction of the input reading
  - In our case always L, i.e. from left to right
- Y – kind of derivation
  - L – left derivation
  - R – right derivation
- P – prefix
  - Subtle division of some grammar classes
- k – look-ahead
  - An integer, usually 1, can be 0 or more generally k
- Examples
  - LL(1), LR(0), LR(1), LL(k), SLR(1), LALR(1)

Expanding definition of FIRST and FOLLOW on k

- If α is a string from grammar symbols, then $FIRST_k(\alpha)$ is a set of terminal words with maximal length k, which are on the beginning of at least one string derived from α. If α can be derived on Λ, then Λ is in $FIRST_k(\alpha)$.
- $FOLLOW_k(A)$ for nonterminal A is a set of terminal words with maximal length k, which can be on the right side of A in any string derived from the start nonterminal ($S \Rightarrow^* \alpha A u \beta$ for some α and β). If A is the right-most symbol in any sentential form, then $ is in $FOLLOW_k(A)$.

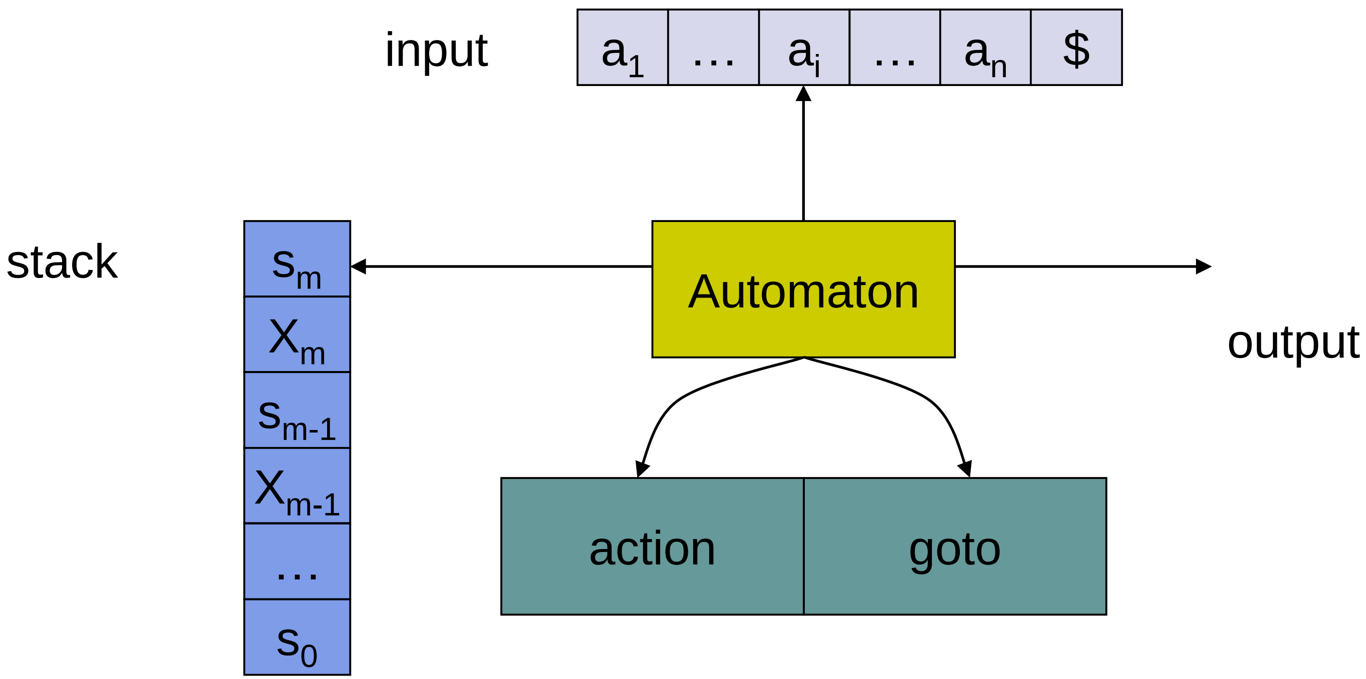
LL(k) grammar

- Context-free grammar $G=(T,N,S,P)$ is a strong LL(k) grammar for $k\geq 1$, if and only if whenever $A\rightarrow\alpha$, $A\rightarrow\beta \in P$ are two distinct ($\alpha\neq\beta$) productions and we have any left sentential forms $uA\gamma$, $vA\delta$, where $u,v\in T^*$ and $\gamma,\delta\in(T\cup N)^*$, the following condition holds:
  - $FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\delta) = \emptyset$.
- LL(k) (not strong)
  - $u=v$, $\gamma=\delta$

Bottom-up analysis

- Attempts to find in reverse the rightmost derivation for an input string
- Attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root
- Replace a substring corresponding to a right side of a production by a nonterminal from the left side of the production in each reduce step
- Used in parser generators
  - Bison – LALR(1), GLR(1)
- Advantages against LL(1) parsers
  - It can be implemented with the same efficiency as top-down parsing
  - The class of decidable languages LR(1) is a proper superset of LL(1)
- SLR(1), LR(1), LALR(1)

LR parser automaton



- s_i are states
  - A state on the top of the stack is the current state of the automaton
- x_i are grammar symbols

LR(1) automaton behavior

- Initial configuration
  - Input pointer points to the first terminal in the input string
  - Initial state s_0 is on the stack
- In each step address table action[s_m, a_i] using s_m and a_i
  - Shift s, where s is a new state
    - It shifts the input tape by 1 terminal and add a_i and s on the top of the stack
  - Reduce using production $A \rightarrow \alpha$
    - Remove r=|α| pairs (s_k, X_k) from the top of the stack, add A on the top of the stack and then goto[s_{m-r}, A] (s_{m-r} is a state on the top of the stack after erasing pairs)
    - Generate an output
  - Accept
    - The input string is accepted
    - Generate an output
  - Error
    - The input string is not in the input language

LR automaton tables for our grammar

state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example of LR parser behavior

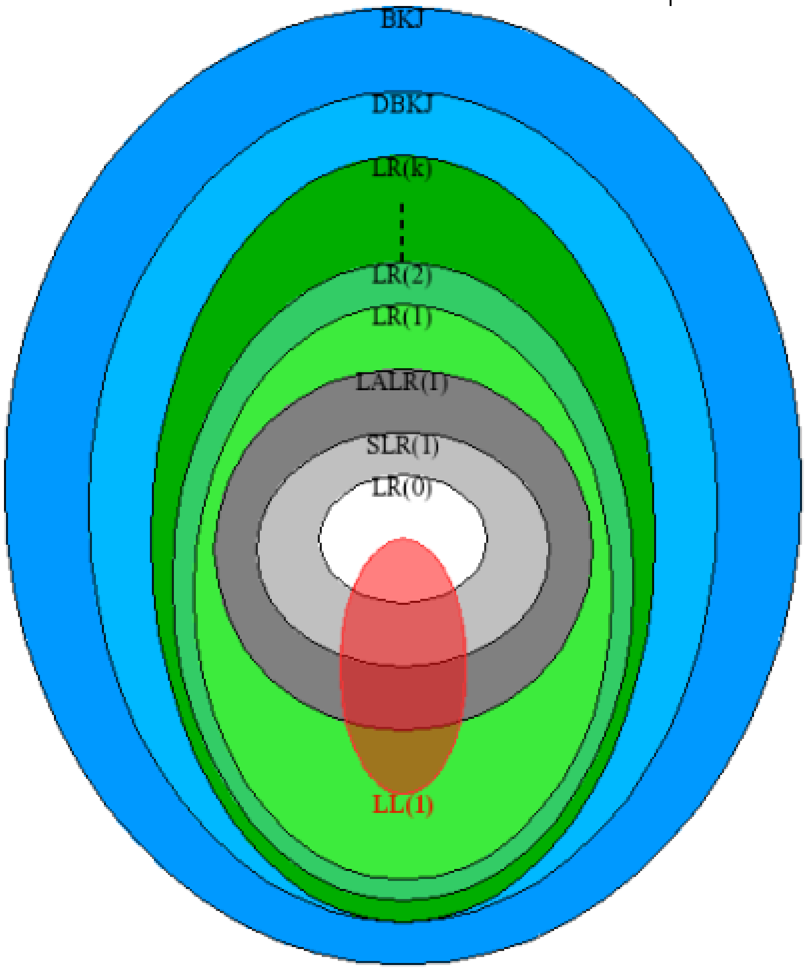
Stack	Input	Action
0	**id+id*id$**	s5
0 **id** 5	**+id*id$**	r6: F→**id**
0 F 3	**+id*id$**	r4: T→F
0 T 2	**+id*id$**	r2: E→T
0 E 1	**+id*id$**	s6
0 E 1 **+** 6	**id*id$**	s5
0 E 1 **+** 6 **id** 5	***id$**	r6: F→**id**
0 E 1 **+** 6 F 3	***id$**	r4: T→F
0 E 1 **+** 6 T 9	***id$**	s7
0 E 1 **+** 6 T 9 ***** 7	**id$**	s5
0 E 1 **+** 6 T 9 ***** 7 **id** 5	**$**	r6: F→**id**
0 E 1 **+** 6 T 9 ***** 7 F 10	**$**	r3: T→T ***** F
0 E 1 **+** 6 T 9	**$**	r1: E→E **+** T
0 E 1	**$**	acc

LR(k) grammar

- Context-free grammar $G=(T,N,S,P)$ is LR(k) grammar for $k\geq 1$, if and only if whenever $A\rightarrow\alpha$, $A\rightarrow\beta \in P$ are two distinct $(\alpha\neq\beta)$ productions of G and we have any two right sentential forms γAu, δAv, where $u,v\in T^*$ and $\gamma,\delta\in(T\cup N)^*$, the following condition holds:
  - $FIRST_k(u) \cap FIRST_k(v) = \emptyset$

Grammars (languages) strength

- Union of all LR(k) are deterministic context-free languages (DBKJ) and it is a proper subset of all context-free languages (BKJ)



Grammar augmentation

- Augmentation of a grammar $G=(T,N,S,P)$ is a grammar $G'=(T,N',S',P')$, where $N'=N\cup\{S'\}$ and $P'=P\cup\{S'\to S\}$
- The augmentation is not necessary whenever S is on the left side of one production and it isn't on any right side of grammar productions
- It helps recognize the end of parsing
- For our grammar:
  - $S'\to E$

LR(0) items

- LR(0) item of a grammar G is a production with a special symbol dot on the right side
  - Special symbol is a valid symbol for comparison of two LR(0) items of a same production. LR(0) items of the same production are different, whenever the dot is on different position. Moreover, the dot is not a grammar symbol
- An example for production $E \rightarrow E + T$:

$E \rightarrow \blacklozenge E + T$

$E \rightarrow E \blacklozenge + T$

$E \rightarrow E + \blacklozenge T$

$E \rightarrow E + T \blacklozenge$

The closure operation

- If I is a set of LR(0) items for a grammar G, then CLOSURE(I) is a set of LR(0) items constructed from I by following rules:
  - Add I to the CLOSURE(I)
  - $\forall$ $A \rightarrow \alpha \blacklozenge B\beta \in$ CLOSURE(I), where $B \in N$, add $\forall$ $B \rightarrow \gamma \in P$ to CLOSURE(I) LR(0) item $B \rightarrow \blacklozenge\gamma$, if it is not already there. Apply this rule until no more new LR(0) items can be added to CLOSURE(I)

Example of closure for our grammar

- I={S’→♦E}
- CLOSURE(I)=
  - S’→ ♦E
  - E → ♦E **+** T
  - E → ♦T
  - T → ♦T ***** F
  - T → ♦F
  - F → ♦**(** E **)**
  - F → ♦**id**

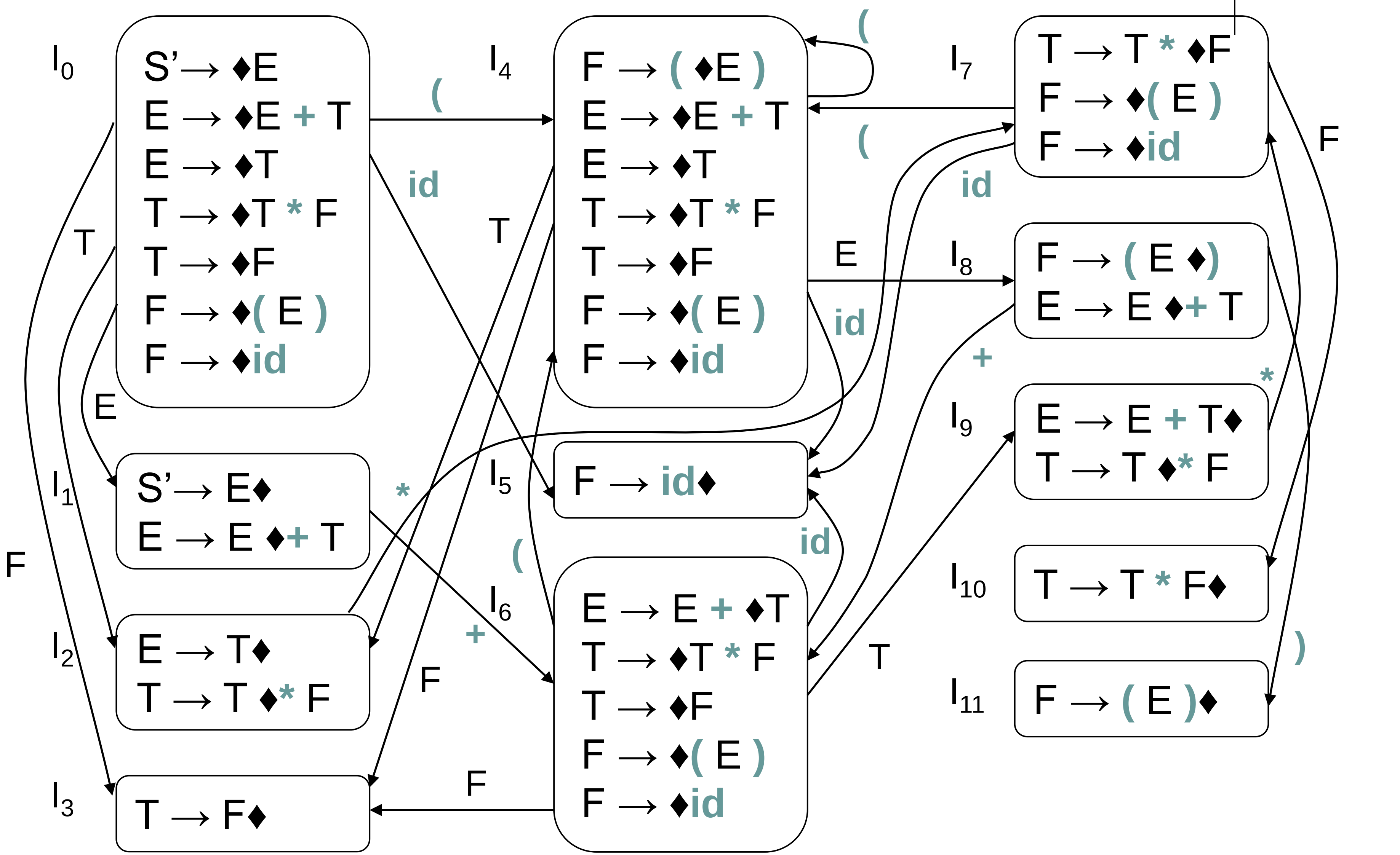
GOTO operation

- GOTO(I, X) operation for a set I of LR(0) items and a grammar symbol X is defined to be the closure of the set of all LR(0) items $A \to \alpha X \blacklozenge \beta$ such that $A \to \alpha \blacklozenge X \beta \in I$

Construction of canonical collection of sets of LR(0) items

- We have an augmented grammar G'=(T,N',S',P')
- Construction of canonical collection C of sets of LR(0) items:
  - We start with C={ CLOSURE({S'→♦S}) }
  - $\forall$ I$\in$C and $\forall$ X$\in$T$\cup$N' such as GOTO(I, X)$\notin$C $\wedge$ GOTO(I, X)$\neq\emptyset$, add GOTO(I, X) to C. Repeat this step, until something new is added to C.

Construction of canonical collection for our grammar



Valid items

- LR(0) item $A \to \beta_1 \blacklozenge \beta_2$ is a valid item for a viable prefix $\alpha\beta_1$, if there is a rightmost derivation $S' \Rightarrow^{+} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$
- It is a great hint for a parser. It helps to decide, if the parser should make a shift or a reduction, if $\alpha\beta_1$ is on top of the stack
- Basic LR parsing theorem: A set of valid items for a viable prefix γ is exactly a set of items reachable from the initial state through the prefix γ by deterministic finite automaton constructed from canonical collection with GOTO transitions.

SLR(1) automaton construction

- We have an augmented grammar G'. Tables of SLR(1) automaton are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(0) items
  - State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows
    - $A \rightarrow \alpha \blacklozenge a\beta \in I_i$, $a \in T \wedge GOTO(I_i,a)=I_j$, then action[i,a]=shift j
    - $A \rightarrow \alpha \blacklozenge \in I_i$, then $\forall a \in FOLLOW(A) \wedge A \neq S'$ is action[i,a]=reduce $A \rightarrow \alpha$
    - $S' \rightarrow S \blacklozenge \in I_i$, then action[i,\$]=accept
  - If there is a conflict in the previous step, the grammar is not a SLR(1) grammar and the automaton cannot be constructed
  - Table goto is indexed by state *i* and $A \in N'$: whenever $GOTO(I_i,A)=I_j$, then goto [i,A]=j
  - All empty cells are filled by error instruction
  - The initial state of the parser is the state, which contains LR(0) item $S' \rightarrow \blacklozenge S$

Full LR(1) automata

- action[i,a] is set to reduction $A \rightarrow \alpha$, when $A \rightarrow \alpha\blacklozenge \in I_i$, $\forall a \in FOLLOW(A)$ for a state *i* during SLR(1) construction
- In some situation, when *i* is on top of the stack, the viable prefix $\beta\alpha$ is in form, where βA cannot be followed by a terminal **a** in any right sentential form. Therefore reduction $A \rightarrow \alpha$ is for lookahed **a** invalid.
- Solution: add more information to states, so we can avoid invalid reductions.

LR(1) items

- The added information is stored as an additional terminal for each LR(0) item. Such item has a form $[A \rightarrow \alpha\blacklozenge\beta, a]$, where $A \rightarrow \alpha\beta \in P$, $a \in T$, and we call it LR(1) item. The terminal **a** is called lookahead.
  - The lookahed has no meaning for $A \rightarrow \alpha\blacklozenge\beta$, where $\beta \neq \Lambda$
  - Reduction $A \rightarrow \alpha$ is set only when $[A \rightarrow \alpha\blacklozenge, a] \in I_i$ for current state *i* and a terminal **a** on the input
  - A set of terminals created from lokaheads of LR(1) items $\subseteq$FOLLOW(A)
- LR(1) item $[A \rightarrow \alpha\blacklozenge\beta, a]$ is valid for viable prefix γ, whenever $\exists$ right derivation $S \Rightarrow^{+} \delta A w \Rightarrow \delta\alpha\beta w$, where
  - $\gamma = \delta\alpha$
  - Either **a** is the first symbol of w or $w = \Lambda$ and **a** is \$

Closure for LR(1) items

- We have a set of LR(1) items I for a grammar G. We define CLOSURE1(I) as a set of LR(1) items constructed from I by following procedure:
  - Add set I to CLOSURE1(I)
  - $\forall\ [A \rightarrow \alpha \blacklozenge B\beta, a] \in$ CLOSURE1(I), where $B \in N$, add LR(1) item $[B \rightarrow \blacklozenge\gamma, b]\ \forall\ B \rightarrow \gamma \in P$ and $\forall b \in FIRST(\beta a)$ to CLOSURE1(I), if it isn't there already. Repeat this step, until something is added to CLOSURE1(I).

GOTO operation for LR(1) items

- We define GOTO1(I, X) operation for a set I of LR(1) items and a grammar symbol X as a CLOSURE1 of a set of all items $[A \rightarrow \alpha X \blacklozenge \beta, a]$ where $[A \rightarrow \alpha \blacklozenge X \beta, a] \in I$

Construction of canonical collection of sets of LR(1) items

- We have an augmented grammar G’=(T,N’,S’,P’)
- Construction of canonical collection C of LR(1) items:
  - We start with C={ CLOSURE1({[S’→♦S,$]}) }
  - Add GOTO1(I, X) to C $\forall$ I$\in$C and $\forall$ X$\in$T$\cup$N’, where GOTO1(I, X)$\notin$C $\wedge$ GOTO1(I, X)$\neq\emptyset$. Repeat this step, until something new is added to C.

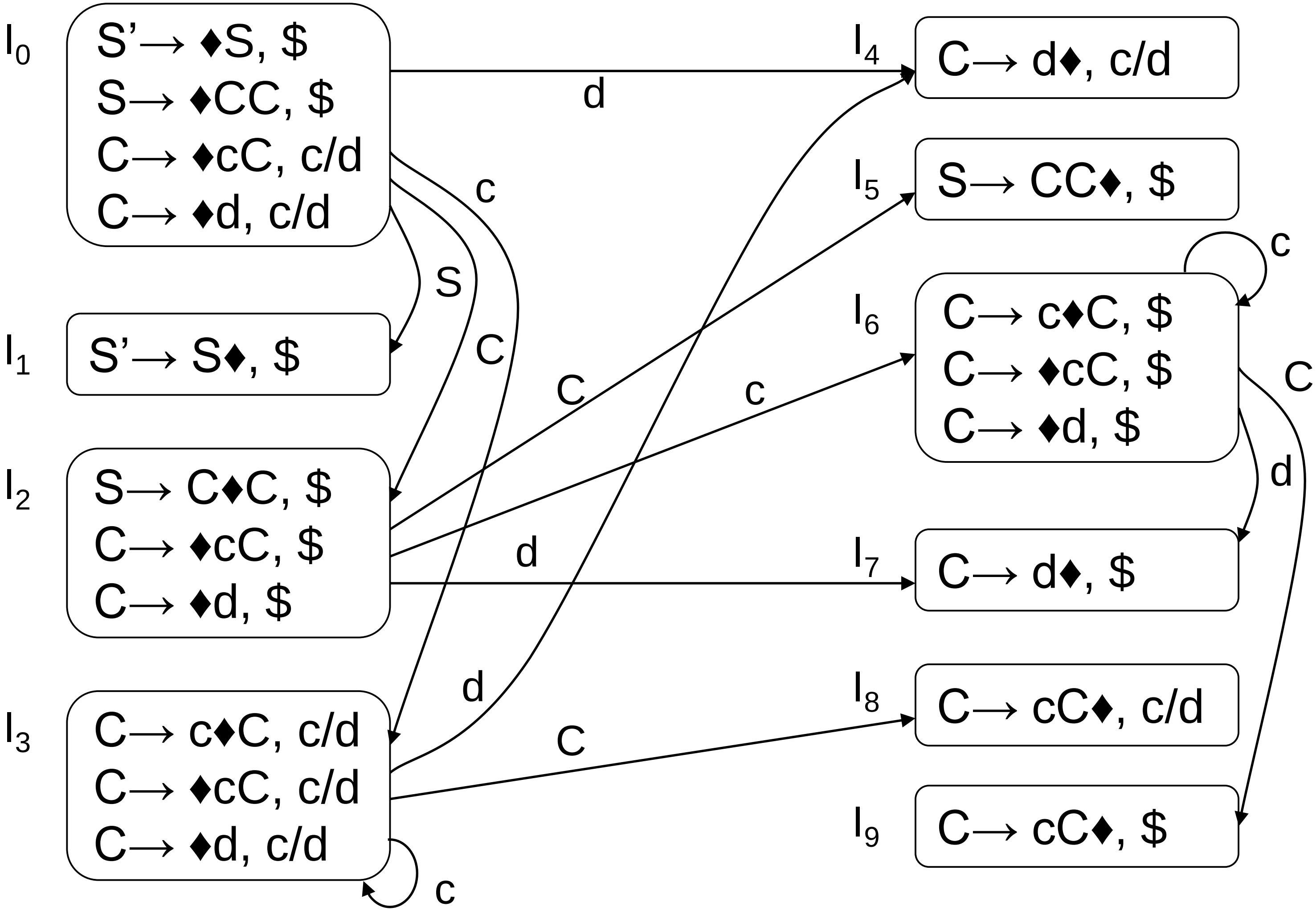
Example of LR(1) grammar, which is not SLR(1)

- $S' \rightarrow S$
- $S \rightarrow CC$
- $C \rightarrow cC$
- $C \rightarrow d$

Example of closure construction for LR(1) items

- $I=\{[S' \rightarrow \blacklozenge S, \$]\}$
- CLOSURE1(I)=
  - $S' \rightarrow \blacklozenge S,\ \$$ $\quad \beta=\Lambda$, FIRST($\beta\$$)=FIRST($\$$)=$\{\$\}$
  - $S \rightarrow \blacklozenge CC,\ \$$ $\quad \beta=C$, FIRST($C\$$)=$\{c,d\}$
  - $C \rightarrow \blacklozenge cC,\ c/d$
  - $C \rightarrow \blacklozenge d,\ c/d$

Example of construction of canonical collection of LR(1) items



LR(1) parser construction

- We have an augmented grammar G'. LR(1) automaton tables are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(1) items
  - State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows
    - $[A \to \alpha \blacklozenge a\beta, b] \in I_i, a \in T \wedge GOTO1(I_i, a) = I_j$, then action[i,a]=shift j
    - $[A \to \alpha \blacklozenge, a] \in I_i \wedge A \neq S'$, then action[i,a]=reduce $A \to \alpha$
    - $[S' \to S \blacklozenge, \$] \in I_i$, then action[i,\$]=accept
  - If there is a conflict in the previous step, the grammar is not a LR(1) grammar and the automaton cannot be constructed
  - Table goto is indexed by state *i* and $A \in N'$: whenever $GOTO1(I_i, A) = I_j$, then goto [i,A]=j
  - All empty cells are filled by error instruction
  - The initial state of the parser is the state, which contains LR(1) item $[S' \to \blacklozenge S, \$]$

LALR

- LALR=LookAhead-LR
- Often used in practice
  - Bison
  - Most common programming languages can be expressed by an LALR grammar
  - Parser tables are considerably smaller then LR(1) tables
- SLR and LALR parsers have the same number of states, LR parsers have greater number of states
  - Common languages have hundreds of states
  - LR(1) parsers have thousands of states for the same grammar

How to make smaller tables?

- Idea: merge sets with the same core into one set including GOTO1 merge
  - Core: a set of LR(0) items (no lookahead)
  - Merge cannot produce shift/reduce conflict
    - Suppose in the union there is a conflict on lookahead **a** for LR(1) items $[A \to \alpha\blacklozenge, a]$ and $[B \to \beta\blacklozenge a\gamma, b]$
    - Cores are same, therefore in the set with $[A \to \alpha\blacklozenge, a]$ must be $[B \to \beta\blacklozenge a\gamma, c]$ as well for some **c**. There was already a shift/reduce conflict before merge
  - Merge can produce reduce/reduce conflict

Easy LALR(1) table construction

- We have an augmented grammar G'. LALR(1) automaton tables are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(1) items
  - For each core in collection C, find all sets having that core, and replace these sets by their union
  - Let C'={ J_0, J_1, …, J_m } be the resulting collection of LR(1) items
  - Table action is constructed for C' in the same manner as for full LR(1) parser
  - If there is a conflict, the grammar is not LALR(1) grammar
  - If J∈C' is the union of sets of LR(1) items I_i ($J=I_1 \cup I_2 \cup \ldots I_k$), then cores GOTO1($I_1$,X), …, GOTO1($I_k$,X) are the same, since I_1, …, I_k all have the same core. Let K be the union of all sets of items having the same core as goto(I_1,X). Then GOTO1(J,X)=K
- Important disadvantage – we need to construct full LR(1)

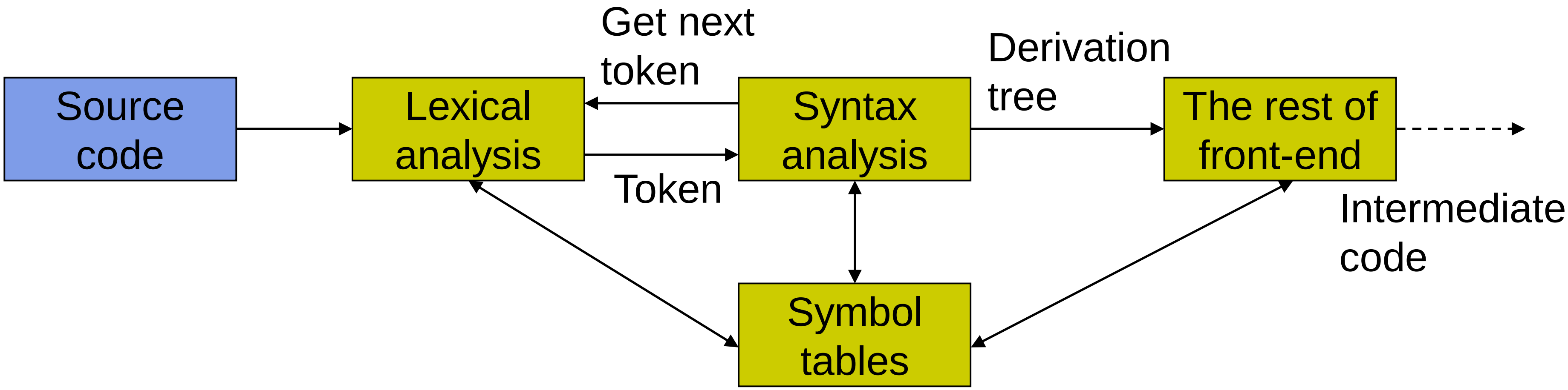
Compiler principles

Syntax analysis

Jakub Yaghob

Syntax analysis

- The main task
  - Decide, whether an input word is a word from an input language
- Other important tasks
  - Syntax-directed translation is the main loop of the compiler
  - Build the derivation tree
- Automaton type
  - We are talking about (deterministic) context-free grammars, therefore we are using (deterministic) pushdown automata



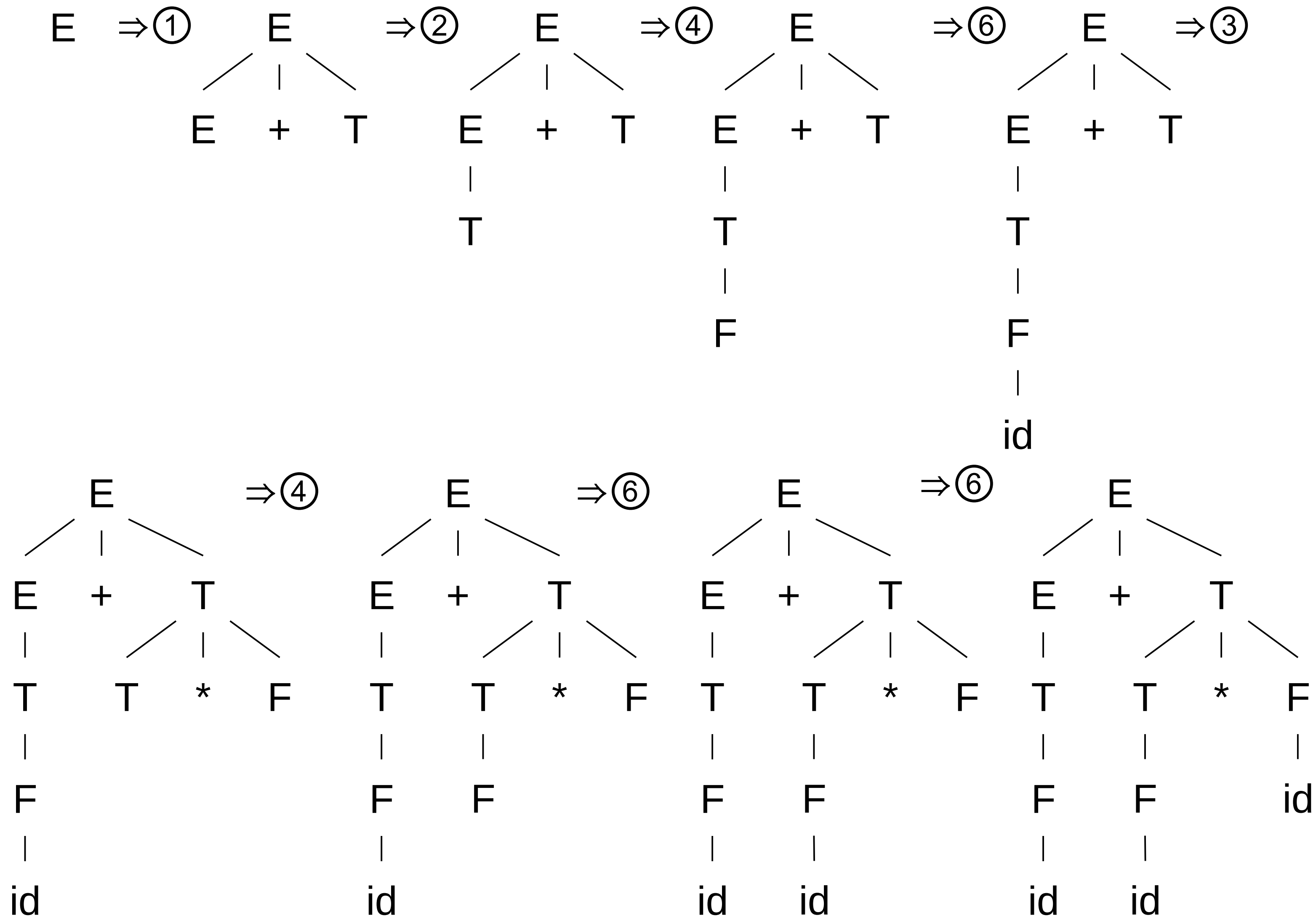
Our grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \mathbf{id}$

Derivation (parse, syntax) tree

- Graphical representation of derivations using trees
  - Vertices are both non-terminals and terminals
  - Edges from inner vertex representing a non-terminal on the left side of a production rule to all symbols from the right side of a production rule
- E ⇒① E+T ⇒② T+T ⇒④ F+T ⇒⑥ id+T ⇒③ id+T*F ⇒④ id+F*F ⇒⑥ id+id*F ⇒⑥ id+id*id

Example


E ⇒① E ⇒② E ⇒④ E ⇒⑥ E ⇒③
E + T E + T E + T E + T
T T T
F F
id
E ⇒④ E ⇒⑥ E ⇒⑥ E
E + T E + T E + T E + T
T T * F T T * F T T * F T T * F
F F F F F F F id
id id id id id id

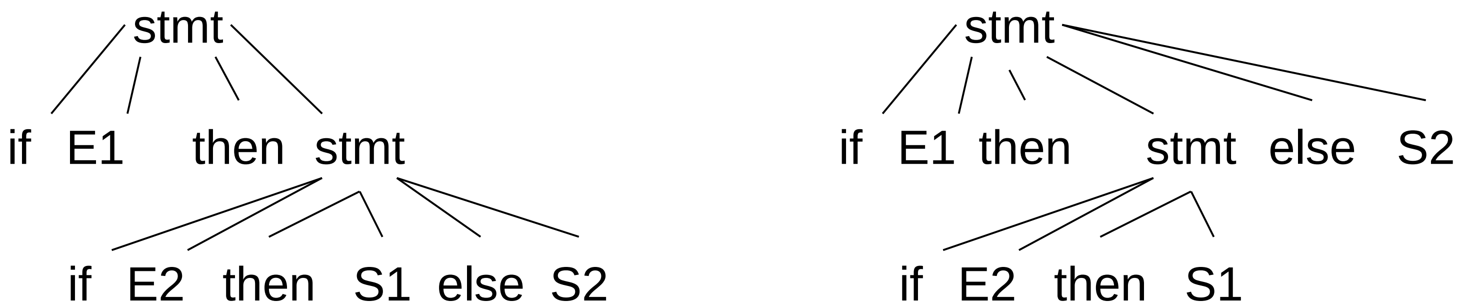
Ambiguous grammar

- We can construct distinct derivation trees for the same input word
- Real-life example (dangling else):
  - stmt → **if** expr **then** stmt

    | **if** expr **then** stmt **else** stmt

    | **while** expr **do** stmt

    | **goto** num
  - Input word: **if** E_1 **then if** E_2 **then** S_1 **else** S_2



Disambiguation

- Clarify, which tree is the right one
- In our case: **else** pairs with nearest “free” **if** (without **else**)
- Idea: “paired” statement is always between **if** and **else**
  - stmt → m_stmt
  - | u_stmt
  - m_stmt → **if** expr **then** m_stmt **else** m_stmt
    | **while** expr **do** m_stmt
    | **goto** num
  - u_stmt → **if** expr **then** stmt
    | **if** expr **then** m_stmt **else** u_stmt
    | **while** expr **do** u_stmt

Left recursion elimination

- A grammar is a left-recursive grammar, when there is a non-terminal A for which it is true that $A \Rightarrow^{+} A\alpha$ for a string α
- It is a problem for top-down parsing
- A simple solution for $\beta\alpha^{m}$:
  - $A \rightarrow A\alpha$
  - $A \rightarrow \beta$
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A'$
  - $A' \rightarrow \Lambda$

Removing left recursion from our grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

1. $E \rightarrow TE'$
2. $E' \rightarrow + TE'$
3. $E' \rightarrow \Lambda$
4. $T \rightarrow FT'$
5. $T' \rightarrow * FT'$
6. $T' \rightarrow \Lambda$
7. $F \rightarrow (E)$
8. $F \rightarrow id$

Left factoring

- It is not clear, which option we should choose

  - $A \rightarrow \alpha\beta_1$
  - $A \rightarrow \alpha\beta_2$

  - $A \rightarrow \alpha A'$
  - $A' \rightarrow \beta_1$
  - $A' \rightarrow \beta_2$

Non-context-free language constructions

- L_1={ wcw | w=(a|b)* }
  - Check, whether an identifier **w** is declared before using
- $L_2=\{ a^n b^m c^n d^m \mid n \geq 1, m \geq 1 \}$
  - Check, whether number of parameters in function call confirms to the function declaration
- $L_3=\{ a^n b^n c^n \mid n \geq 0 \}$
  - The problem of “underscoring” a word
    - **a** is a char, **b** is BS, **c** is underscore
  - (abc)* is a regular expression

Operators FIRST and FOLLOW – definitions

- If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If α can be derived to Λ, then Λ is also in FIRST(α)
- Define FOLLOW(A), for nonterminal A, to be the set of terminals that can appear immediately to the right of A in some string, where exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β. If A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A).

Construction of the FIRST operator

- Construction for a grammar symbol X
  - If X is terminal, then FIRST(X)={X}
  - If $X \rightarrow \Lambda$ is a production, then add Λ to FIRST(X)
  - If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place **a** in FIRST(X), if for some i, **a** is in FIRST(Y_i) and $\Lambda \in$FIRST(Y_j) $\forall$ j<i. If $\Lambda \in$FIRST(Y_j) $\forall$ j, then add Λ to FIRST(X)
- Construction for any string
  - The construction of the FIRST operator for a string $X_1 X_2 \ldots X_n$ is similar as for nonterminal.

Construction of the FOLLOW operator

- Construction for a nonterminal A
  - Place \$ in FOLLOW(S), where S is the start symbol of a grammar and \$ is EOS
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for Λ is placed in FOLLOW(B)
  - If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where $\Lambda \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW – an example for our grammar

- FIRST(E)={ **(**, **id** }
- FIRST(T)={ **(**, **id** }
- FIRST(F)={ **(**, **id** }
- FIRST(E’)={ **+**, Λ }
- FIRST(T’)={ *****, Λ }

- FOLLOW(E)={ **)**, $ }
- FOLLOW(E’)={ **)**, $ }
- FOLLOW(T)={ **+**,**)**, $ }
- FOLLOW(T’)={ **+**,**)**, $ }
- FOLLOW(F)={ **+**, *****, **)**, $ }

Top-down parsing

- An attempt to find a leftmost derivation for an input string
- An attempt to construct a parse tree for the input starting from the root and creating the nodes of the tree in preorder
- Recursive-descent parsing
  - Recursive descent using procedures
- Nonrecursive predictive parsing
  - An automaton with an explicit stack
- Both solutions have a problem with left recursion in a grammar
- Many current parser generators use top-down parsing
  - ANTLR, CocoR – LL(1) grammars with conflict resolution using dynamic look-ahead expansion to LL(k)

Recursive-descent parsing

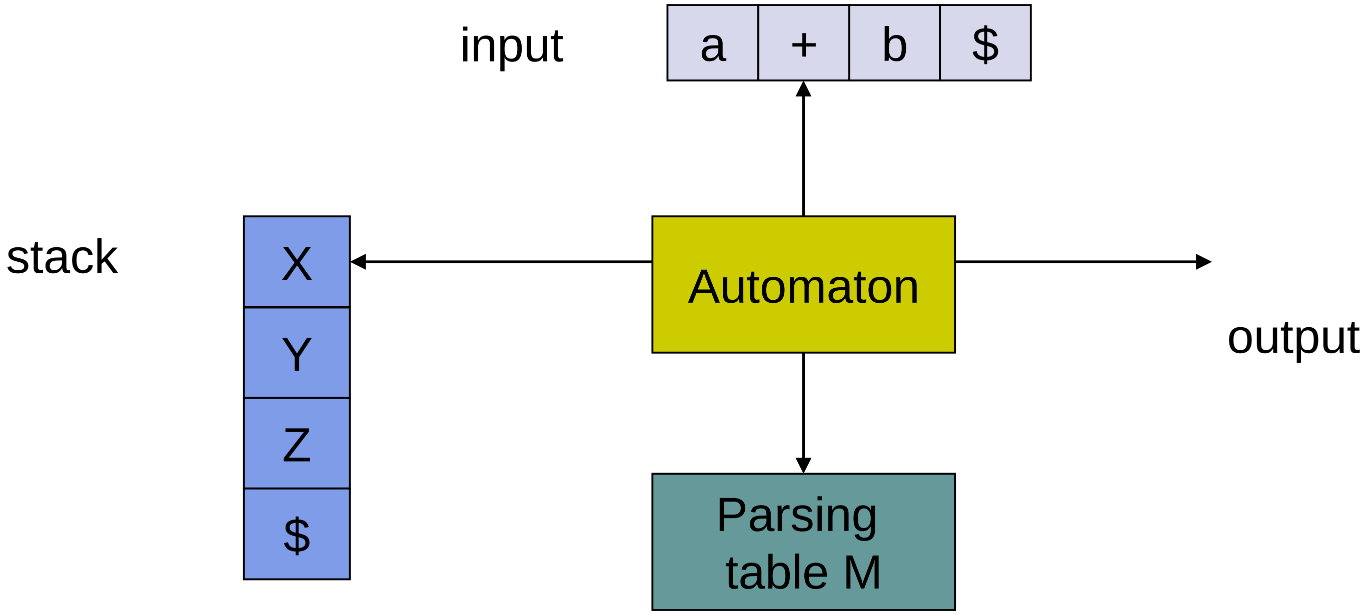
- One procedure/function for each nonterminal of a grammar
- Each procedure does two things
  - It decides, which grammar production with given nonterminal on the left side will be used using look-ahead. A production with right side α will be used, when the look-ahead is in FIRST(α). If there is a conflict for the look-ahead among some production right sides, the grammar is not suitable for recursive-descent parsing. A production with Λ on the right side will be used, if the look-ahead is not in FIRST of any right side.
  - Procedure code copies the right side of a production. Nonterminal means calling a procedure for this nonterminal. Terminal is compared with the look-ahead. If they are equal, a next terminal is read. If they are not equal, it is an error.

Recursive-descent parsing – example for our grammar

```
void match(token t) {
  if(lookahead==t)
    lookahead = nexttoken();
  else error();
}
void E(void) {
  T(); Eap();
}
void Eap(void) {
  if(lookahead=='+') {
    match('+'); T(); Eap();
  }
}
void T(void) {
  F(); Tap();
}
```

```
void Tap(void) {
  if(lookahead=='*') {
    match('*'); F(); Tap(); }
}
void F(void) {
  switch(lookahead) {
    case '(':match('('); E();
             match(')');break;
    case 'id':
      match('id'); break;
    default:
      error();
  }
}
```

Nonrecursive predictive parsing



- Parsing table M[A, a], where A is nonterminal and **a** is terminal
- The stack contains grammar symbols

LL(1) automaton behavior

- Initial configuration
  - Input pointer points to the first terminal in the input string
  - The stack contains the start symbol of the grammar on top of $
- In each step, the automaton decides, what to do, using a symbol X on top of the stack and a terminal **a**, pointed by input pointer
  - If X=a=$, the parser halts, parsing finished successfully
  - If X=a≠$, the parser pops X from the stack and advances the input pointer to the next input symbol
  - If X≠a and X∈T, the parser reports error
  - If X is a nonterminal, the parser uses entry M[X, a]. If this entry is a production, the parser replaces X on top of the stack by the right side (leftmost symbol on top of the stack). At the same time, the parser generates an output about using the production. If the entry is **error**, the parser informs about a syntax error.

Construction of predictive parsing tables

- For each production $A \to \alpha$ do following steps
  - For $\forall\ a \in \mathrm{FIRST}(\alpha)$ add $A \to \alpha$ to $M[A, a]$
  - If $\Lambda \in \mathrm{FIRST}(\alpha)$, add $A \to \alpha$ to $M\ [A, b]$ $\forall\ b \in \mathrm{FOLLOW}(A)$. Moreover, if $\$ \in \mathrm{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, \$]$
- Mark each empty entry in M as **error**

Example of table construction for our grammar

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E’		$E' \rightarrow +TE'$			$E' \rightarrow \Lambda$	$E' \rightarrow \Lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T’		$T' \rightarrow \Lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \Lambda$	$T' \rightarrow \Lambda$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Example of parser behavior for our grammar

Stack	Input	Output
$E	**id+id*id$**	
$E’T	**id+id*id$**	E→TE’
$E’T’F	**id+id*id$**	T→FT’
$E’T’**id**	**id+id*id$**	F→**id**
$E’T’	**+id*id$**	
$E’	**+id*id$**	T’→Λ
$E’T**+**	**+id*id$**	E’→**+**TE’
$E’T	**id*id$**	
$E’T’F	**id*id$**	T→FT’

Stack	Input	Output
$E’T’**id**	**id*id$**	F→**id**
$E’T’	***id$**	
$E’T’F*****	***id$**	T’→*****FT’
$E’T’F	**id$**	
$E’T’**id**	**id$**	F→**id**
$E’T’	**$**	
$E’	**$**	T’→Λ
$	**$**	E’→Λ

LL(1) grammar

- Context-free grammar $G=(T,N,S,P)$ is a LL(1) grammar, if and only if whenever $A\rightarrow\alpha$, $A\rightarrow\beta \in P$ are two distinct ($\alpha\neq\beta$) productions of G and we have any left sentential forms $uA\gamma$, $vA\delta$, where $u,v\in T^*$ and $\gamma,\delta\in(T\cup N)^*$, the following condition holds:
  - $FIRST(\alpha\gamma) \cap FIRST(\beta\delta) = \emptyset$
- Simplified detection: no ambiguous or left-recursive grammar can be LL(1)

Grammar terminology

- PXY(k)
- X – direction of the input reading
    - In our case always L, i.e. from left to right
- Y – kind of derivation
    - L – left derivation
    - R – right derivation
- P – prefix
    - Subtle division of some grammar classes
- k – look-ahead
    - An integer, usually 1, can be 0 or more generally k
- Examples
    - LL(1), LR(0), LR(1), LL(k), SLR(1), LALR(1)

Expanding definition of FIRST and FOLLOW on k

- If α is a string from grammar symbols, then $FIRST_k(\alpha)$ is a set of terminal words with maximal length k, which are on the beginning of at least one string derived from α. If α can be derived on Λ, then Λ is in $FIRST_k(\alpha)$.
- $FOLLOW_k(A)$ for nonterminal A is a set of terminal words with maximal length k, which can be on the right side of A in any string derived from the start nonterminal ($S \Rightarrow^* \alpha A u \beta$ for some α and β). If A is the right-most symbol in any sentential form, then $ is in $FOLLOW_k(A)$.

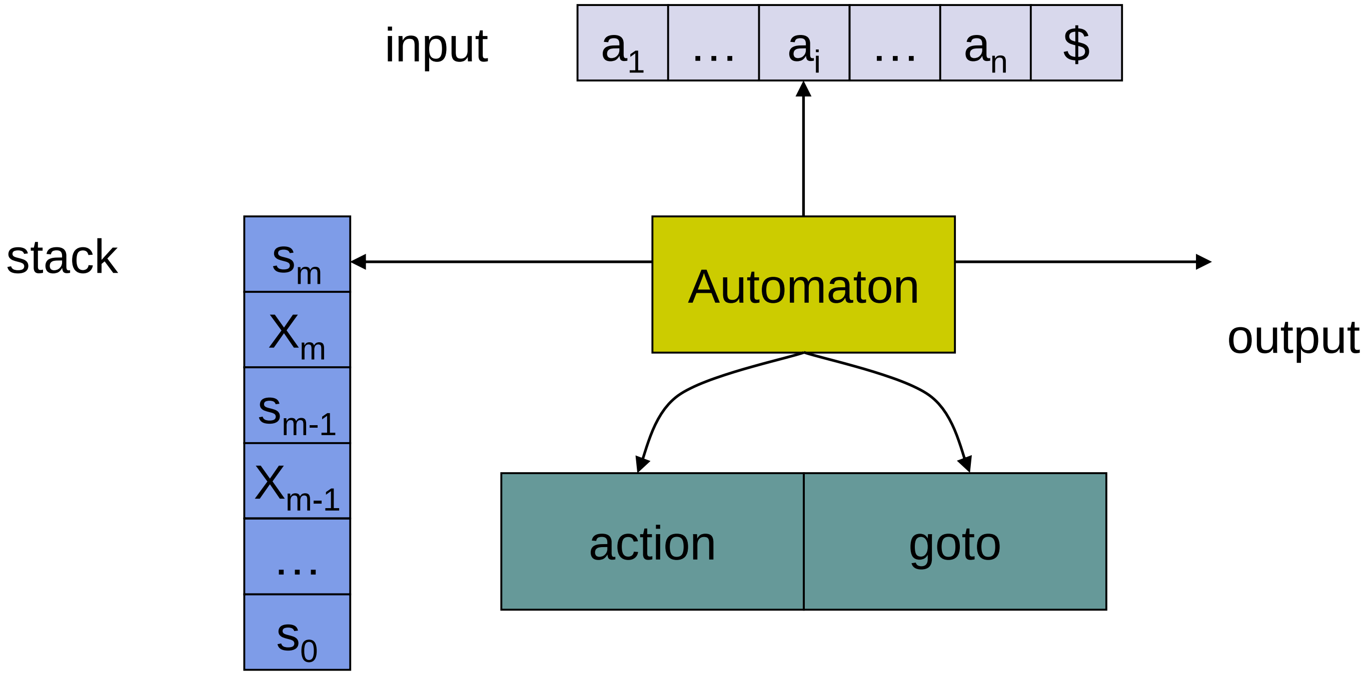
LL(k) grammar

- Context-free grammar $G=(T,N,S,P)$ is a strong LL(k) grammar for $k \geq 1$, if and only if whenever $A \rightarrow \alpha$, $A \rightarrow \beta \in P$ are two distinct ($\alpha \neq \beta$) productions and we have any left sentential forms $uA\gamma$, $vA\delta$, where $u,v \in T^*$ and $\gamma,\delta \in (T \cup N)^*$, the following condition holds:
  - $FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\delta) = \emptyset$.
- LL(k) (not strong)
  - $u=v$, $\gamma=\delta$

Bottom-up analysis

- Attempts to find in reverse the rightmost derivation for an input string
- Attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root
- Replace a substring corresponding to a right side of a production by a nonterminal from the left side of the production in each reduce step
- Used in parser generators
  - Bison – LALR(1), GLR(1)
- Advantages against LL(1) parsers
  - It can be implemented with the same efficiency as top-down parsing
  - The class of decidable languages LR(1) is a proper superset of LL(1)
- SLR(1), LR(1), LALR(1)

LR parser automaton



- s_i are states
  - A state on the top of the stack is the current state of the automaton
- x_i are grammar symbols

LR(1) automaton behavior

- Initial configuration
  - Input pointer points to the first terminal in the input string
  - Initial state s_0 is on the stack
- In each step address table action[s_m, a_i] using s_m and a_i
  - Shift s, where s is a new state
    - It shifts the input tape by 1 terminal and add a_i and s on the top of the stack
  - Reduce using production $A \rightarrow \alpha$
    - Remove $r=|\alpha|$ pairs (s_k, X_k) from the top of the stack, add A on the top of the stack and then goto[s_{m-r}, A] (s_{m-r} is a state on the top of the stack after erasing pairs)
    - Generate an output
  - Accept
    - The input string is accepted
    - Generate an output
  - Error
    - The input string is not in the input language

LR automaton tables for our grammar

state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example of LR parser behavior

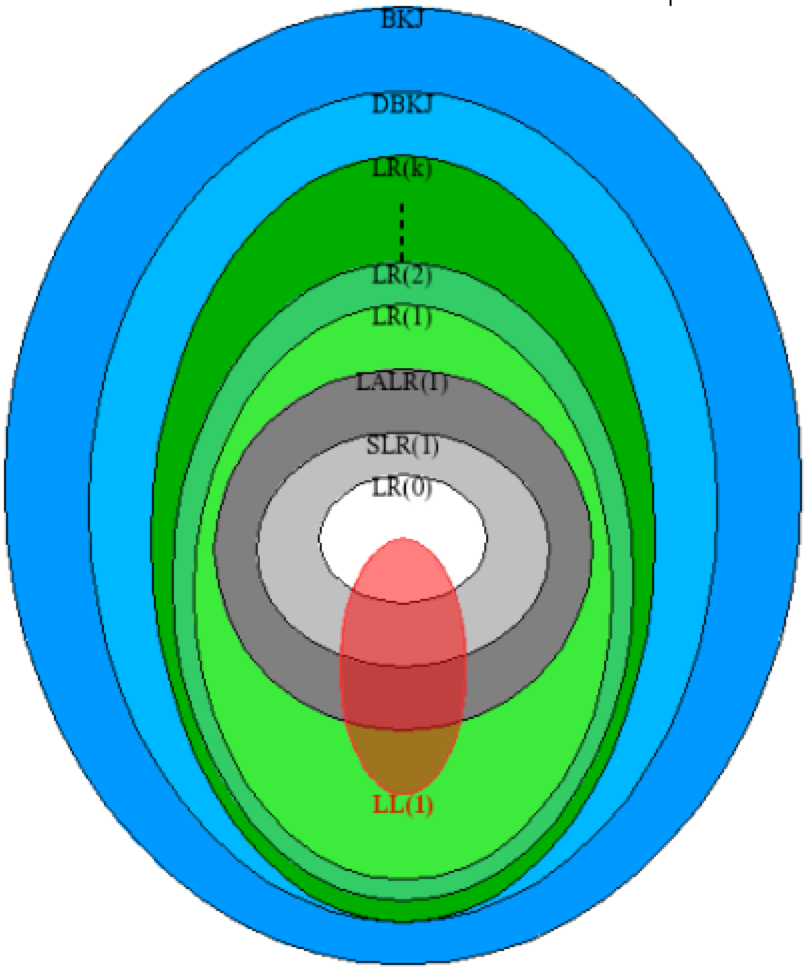
Stack	Input	Action
0	**id+id*id$**	s5
0 **id** 5	**+id*id$**	r6: F→**id**
0 F 3	**+id*id$**	r4: T→F
0 T 2	**+id*id$**	r2: E→T
0 E 1	**+id*id$**	s6
0 E 1 **+** 6	**id*id$**	s5
0 E 1 **+** 6 **id** 5	***id$**	r6: F→**id**
0 E 1 **+** 6 F 3	***id$**	r4: T→F
0 E 1 **+** 6 T 9	***id$**	s7
0 E 1 **+** 6 T 9 ***** 7	**id$**	s5
0 E 1 **+** 6 T 9 ***** 7 **id** 5	**$**	r6: F→**id**
0 E 1 **+** 6 T 9 ***** 7 F 10	**$**	r3: T→T ***** F
0 E 1 **+** 6 T 9	**$**	r1: E→E **+** T
0 E 1	**$**	acc

LR(k) grammar

- Context-free grammar $G=(T,N,S,P)$ is LR(k) grammar for $k \geq 1$, if and only if whenever $A \longrightarrow \alpha$, $A \longrightarrow \beta \in P$ are two distinct ($\alpha \neq \beta$) productions of G and we have any two right sentential forms $\gamma A u$, $\delta A v$, where $u,v \in T^*$ and $\gamma,\delta \in (T \cup N)^*$, the following condition holds:
  - $FIRST_k(u) \cap FIRST_k(v) = \emptyset$

Grammars (languages) strength

- Union of all LR(k) are deterministic context-free languages (DBKJ) and it is a proper subset of all context-free languages (BKJ)



Grammar augmentation

- Augmentation of a grammar $G=(T,N,S,P)$ is a grammar $G'=(T,N',S',P')$, where $N'=N\cup\{S'\}$ and $P'=P\cup\{S'\to S\}$
- The augmentation is not necessary whenever S is on the left side of one production and it isn’t on any right side of grammar productions
- It helps recognize the end of parsing
- For our grammar:
  - $S'\to E$

LR(0) items

- LR(0) item of a grammar G is a production with a special symbol dot on the right side
  - Special symbol is a valid symbol for comparison of two LR(0) items of a same production. LR(0) items of the same production are different, whenever the dot is on different position. Moreover, the dot is not a grammar symbol
- An example for production $E \to E + T$:

$E \to \blacklozenge E + T$ $\qquad$ $E \to E + \blacklozenge T$

$E \to E \blacklozenge + T$ $\qquad$ $E \to E + T \blacklozenge$

The closure operation

- If I is a set of LR(0) items for a grammar G, then CLOSURE(I) is a set of LR(0) items constructed from I by following rules:
  - Add I to the CLOSURE(I)
  - $\forall\ A \rightarrow \alpha \blacklozenge B\beta \in$ CLOSURE(I), where $B \in N$, add $\forall\ B \rightarrow \gamma \in P$ to CLOSURE(I) LR(0) item $B \rightarrow \blacklozenge\gamma$, if it is not already there. Apply this rule until no more new LR(0) items can be added to CLOSURE(I)

Example of closure for our grammar

- I={S'→♦E}
- CLOSURE(I)=
  - S'→ ♦E
  - E → ♦E **+** T
  - E → ♦T
  - T → ♦T ***** F
  - T → ♦F
  - F → ♦**(** E **)**
  - F → ♦**id**

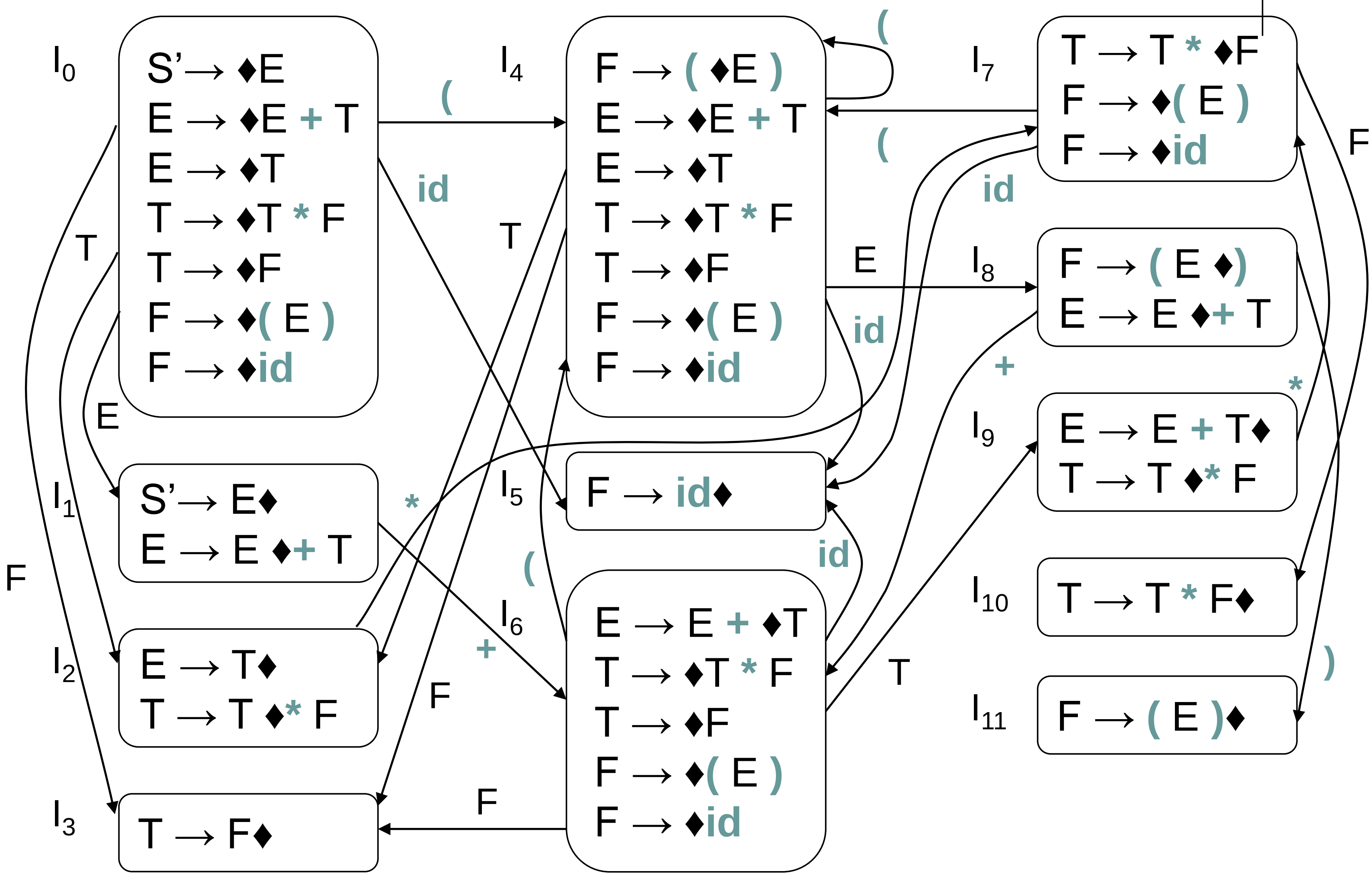
GOTO operation

- GOTO(I, X) operation for a set I of LR(0) items and a grammar symbol X is defined to be the closure of the set of all LR(0) items $A \to \alpha X \blacklozenge \beta$ such that $A \to \alpha \blacklozenge X \beta \in I$

Construction of canonical collection of sets of LR(0) items

- We have an augmented grammar G'=(T,N',S',P')
- Construction of canonical collection C of sets of LR(0) items:
  - We start with C={ CLOSURE({S'→♦S}) }
  - $\forall$ I$\in$C and $\forall$ X$\in$T$\cup$N' such as GOTO(I, X)$\notin$C $\wedge$ GOTO(I, X)$\neq\emptyset$, add GOTO(I, X) to C. Repeat this step, until something new is added to C.

Construction of canonical collection for our grammar



Valid items

- LR(0) item $A \rightarrow \beta_1 \blacklozenge \beta_2$ is a valid item for a viable prefix $\alpha\beta_1$, if there is a rightmost derivation $S' \Rightarrow^{+} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$
- It is a great hint for a parser. It helps to decide, if the parser should make a shift or a reduction, if $\alpha\beta_1$ is on top of the stack
- Basic LR parsing theorem: A set of valid items for a viable prefix γ is exactly a set of items reachable from the initial state through the prefix γ by deterministic finite automaton constructed from canonical collection with GOTO transitions.

SLR(1) automaton construction

- We have an augmented grammar G'. Tables of SLR(1) automaton are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(0) items
  - State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows
    - $A \rightarrow \alpha \blacklozenge a\beta \in I_i$, $a \in T \land \mathrm{GOTO}(I_i,a)=I_j$, then action[i,a]=shift j
    - $A \rightarrow \alpha \blacklozenge \in I_i$, then $\forall a \in \mathrm{FOLLOW}(A) \land A \neq S'$ is action[i,a]=reduce $A \rightarrow \alpha$
    - $S' \rightarrow S \blacklozenge \in I_i$, then action[i,$]=accept
  - If there is a conflict in the previous step, the grammar is not a SLR(1) grammar and the automaton cannot be constructed
  - Table goto is indexed by state *i* and $A \in N'$: whenever $\mathrm{GOTO}(I_i,A)=I_j$, then goto [i,A]=j
  - All empty cells are filled by error instruction
  - The initial state of the parser is the state, which contains LR(0) item $S' \rightarrow \blacklozenge S$

Full LR(1) automata

- action[i,a] is set to reduction $A \rightarrow \alpha$, when $A \rightarrow \alpha \blacklozenge \in I_i$, $\forall a \in FOLLOW(A)$ for a state *i* during SLR(1) construction
- In some situation, when *i* is on top of the stack, the viable prefix $\beta\alpha$ is in form, where βA cannot be followed by a terminal **a** in any right sentential form. Therefore reduction $A \rightarrow \alpha$ is for lookahed **a** invalid.
- Solution: add more information to states, so we can avoid invalid reductions.

LR(1) items

- The added information is stored as an additional terminal for each LR(0) item. Such item has a form $[A\rightarrow\alpha\blacklozenge\beta,a]$, where $A\rightarrow\alpha\beta\in P$, $a\in T$, and we call it LR(1) item. The terminal **a** is called lookahead.
  - The lookahed has no meaning for $A\rightarrow\alpha\blacklozenge\beta$, where $\beta\neq\Lambda$
  - Reduction $A\rightarrow\alpha$ is set only when $[A\rightarrow\alpha\blacklozenge,a]\in I_i$ for current state *i* and a terminal **a** on the input
  - A set of terminals created from lokaheads of LR(1) items $\subseteq$FOLLOW(A)
- LR(1) item $[A\rightarrow\alpha\blacklozenge\beta,a]$ is valid for viable prefix γ, whenever $\exists$ right derivation $S\Rightarrow^{+}\delta Aw\Rightarrow\delta\alpha\beta w$, where
  - $\gamma=\delta\alpha$
  - Either **a** is the first symbol of w or $w=\Lambda$ and **a** is \$

Closure for LR(1) items

- We have a set of LR(1) items I for a grammar G. We define CLOSURE1(I) as a set of LR(1) items constructed from I by following procedure:
  - Add set I to CLOSURE1(I)
  - $\forall$ $[A \rightarrow \alpha \blacklozenge B\beta, a] \in$ CLOSURE1(I), where $B \in N$, add LR(1) item $[B \rightarrow \blacklozenge \gamma, b]$ $\forall$ $B \rightarrow \gamma \in P$ and $\forall b \in$ FIRST(βa) to CLOSURE1(I), if it isn't there already. Repeat this step, until something is added to CLOSURE1(I).

GOTO operation for LR(1) items

- We define GOTO1(I, X) operation for a set I of LR(1) items and a grammar symbol X as a CLOSURE1 of a set of all items $[A \to \alpha X \blacklozenge \beta, a]$ where $[A \to \alpha \blacklozenge X \beta, a] \in I$

Construction of canonical collection of sets of LR(1) items

- We have an augmented grammar G'=(T,N',S',P')
- Construction of canonical collection C of LR(1) items:
  - We start with C={ CLOSURE1({[S'$\rightarrow\blacklozenge$S,$]}) }
  - Add GOTO1(I, X) to C $\forall$ I$\in$C and $\forall$ X$\in$T$\cup$N', where GOTO1(I, X)$\notin$C $\wedge$ GOTO1(I, X)$\neq\emptyset$. Repeat this step, until something new is added to C.

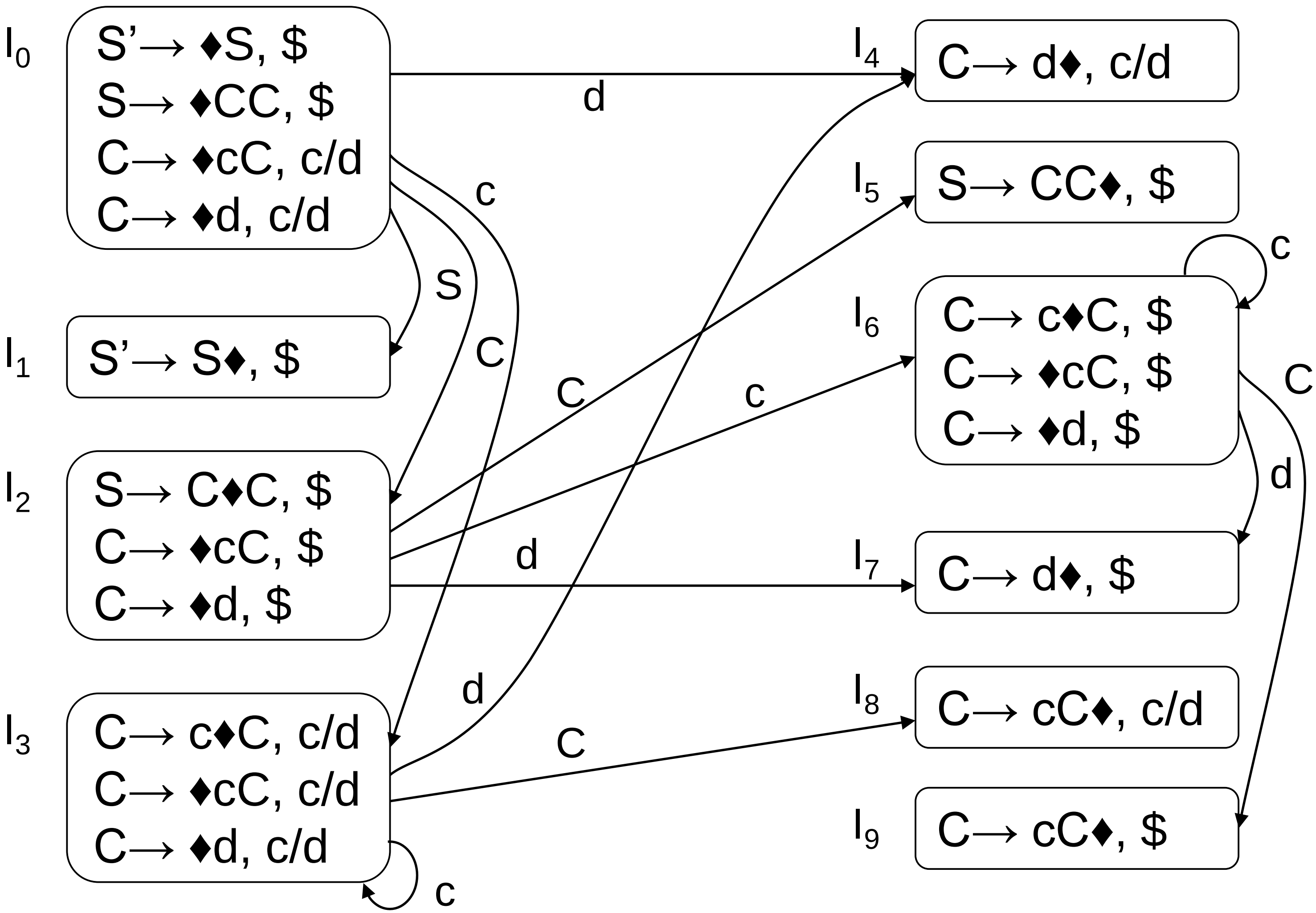
Example of LR(1) grammar, which is not SLR(1)

- $S' \rightarrow S$
- $S \rightarrow CC$
- $C \rightarrow cC$
- $C \rightarrow d$

Example of closure construction for LR(1) items

- I={[S’→♦S,$]}
- CLOSURE1(I)=
  - S’→ ♦S, $ β=Λ,FIRST(β$)=FIRST($)={$}
  - S→ ♦CC, $ β=C,FIRST(C$)={c,d}
  - C→ ♦cC, c/d
  - C→ ♦d, c/d

Example of construction of canonical collection of LR(1) items



LR(1) parser construction

- We have an augmented grammar G'. LR(1) automaton tables are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(1) items
  - State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows
    - $[A \to \alpha \blacklozenge a\beta, b] \in I_i, a \in T \wedge GOTO1(I_i, a) = I_j$, then action[i,a]=shift j
    - $[A \to \alpha \blacklozenge, a] \in I_i \wedge A \neq S'$, then action[i,a]=reduce $A \to \alpha$
    - $[S' \to S \blacklozenge, \$] \in I_i$, then action[i,\$]=accept
  - If there is a conflict in the previous step, the grammar is not a LR(1) grammar and the automaton cannot be constructed
  - Table goto is indexed by state *i* and $A \in N'$: whenever $GOTO1(I_i, A) = I_j$, then goto [i,A]=j
  - All empty cells are filled by error instruction
  - The initial state of the parser is the state, which contains LR(1) item $[S' \to \blacklozenge S, \$]$

LALR

- LALR=LookAhead-LR
- Often used in practice
  - Bison
  - Most common programming languages can be expressed by an LALR grammar
  - Parser tables are considerably smaller then LR(1) tables
- SLR and LALR parsers have the same number of states, LR parsers have greater number of states
  - Common languages have hundreds of states
  - LR(1) parsers have thousands of states for the same grammar

How to make smaller tables?

- Idea: merge sets with the same core into one set including GOTO1 merge
  - Core: a set of LR(0) items (no lookahead)
  - Merge cannot produce shift/reduce conflict
    - Suppose in the union there is a conflict on lookahead **a** for LR(1) items $[A\to\alpha\blacklozenge,a]$ and $[B\to\beta\blacklozenge a\gamma,b]$
    - Cores are same, therefore in the set with $[A\to\alpha\blacklozenge,a]$ must be $[B\to\beta\blacklozenge a\gamma,c]$ as well for some **c**. There was already a shift/reduce conflict before merge
  - Merge can produce reduce/reduce conflict

Easy LALR(1) table construction

- We have an augmented grammar G'. LALR(1) automaton tables are constructed by following algorithm
  - Construct a canonical collection C of sets of LR(1) items
  - For each core in collection C, find all sets having that core, and replace these sets by their union
  - Let C'={ J_0, J_1, …, J_m } be the resulting collection of LR(1) items
  - Table action is constructed for C' in the same manner as for full LR(1) parser
  - If there is a conflict, the grammar is not LALR(1) grammar
  - If $J \in C'$ is the union of sets of LR(1) items I_i ($J=I_1 \cup I_2 \cup \ldots I_k$), then cores GOTO1($I_1$,X), …, GOTO1($I_k$,X) are the same, since I_1, …, I_k all have the same core. Let K be the union of all sets of items having the same core as goto(I_1,X). Then GOTO1(J,X)=K
- Important disadvantage – we need to construct full LR(1)

Compiler principles

Intermediate code

Jakub Yaghob

Intermediate code

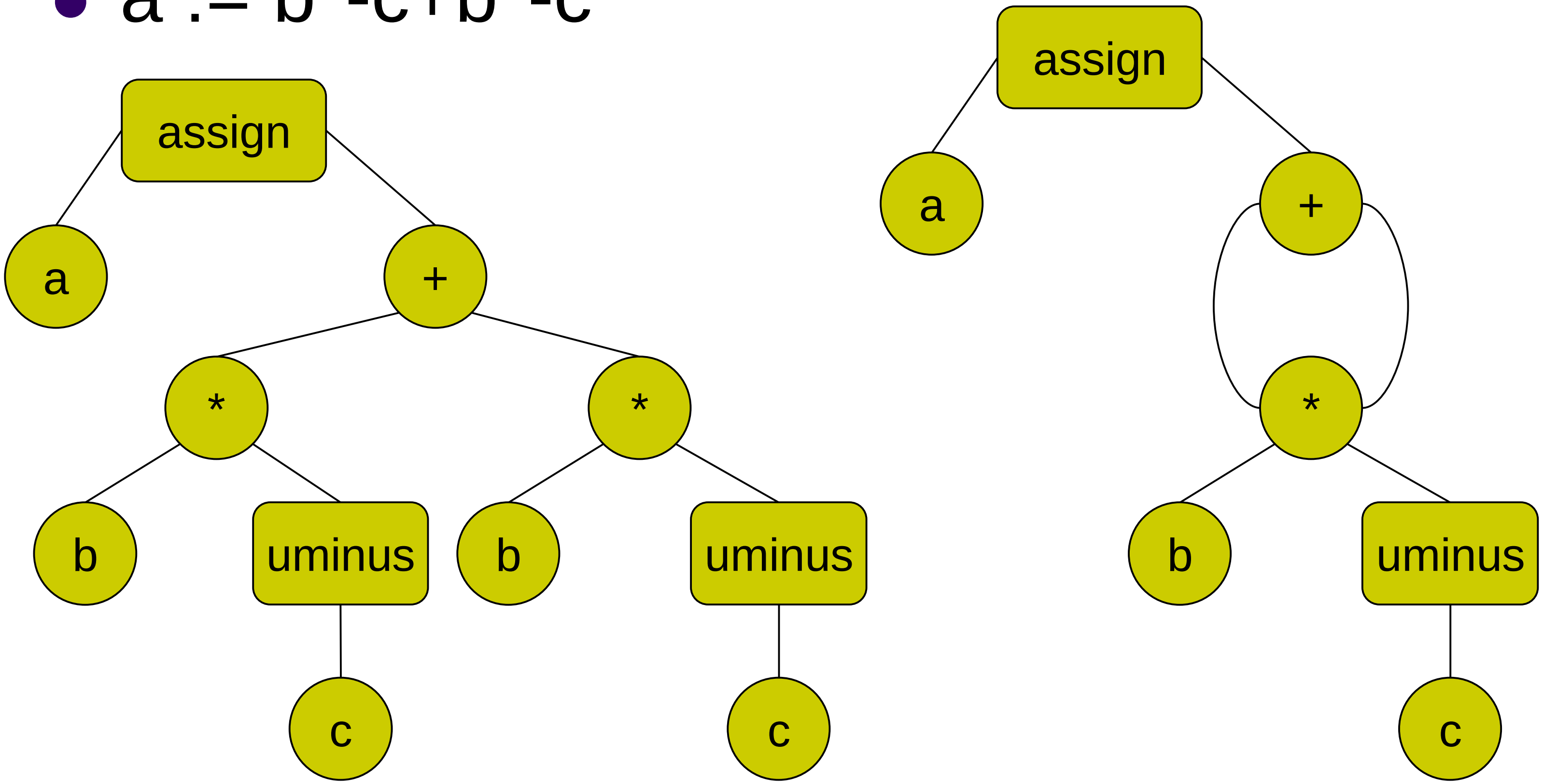
- Intermediate representation of the source code
- Separates front end from back end
- Advantages
  - Different back ends for the same input language – support for different CPU architectures
    - gcc
  - Different front ends for the same output language – support more programming languages for the same CPU architecture
    - .NET
  - A machine-independent optimizations – HLO

Intermediate languages

- Syntax tree
  - Can be even DAG
- Postfix notation
  - Linearized representation of a syntax tree
  - Tree edges not in notation, can be reconstructed from the order and number of operands of an operator
- Three-address code
  - Linearized representation of a syntax tree as well
  - A sequence of statements in form
    - x := y op z

Examples – syntax tree and DAG

- a := b*-c+b*-c



Examples – postfix notation and three-address code

- a b c uminus * b c uminus * + assign

- t1 := -c
- t2 := b * t1
- t3 := -c
- t4 := b * t3
- t5 := t2 + t4
- a := t5

- t1 := -c
- t2 := b * t1
- t5 := t2 + t2
- a := t5

Three-address code operands

- A name
  - A variable
  - A type
  - Other names
- A constant
  - Different literals (UINT, string, …)
- Temporary variable
  - Generated by a compiler
  - Easily they can be thought of as a CPU registers

Types of three-address statements

- Binary arithmetic and logical operation
- Unary operations
- Assignment/copy
- Unconditional jump
- Conditional jump
- Procedure/function call mechanism
  - Parameters, call, return
- Array indexation
- Address operators
  - Address of an object, dereference
- Declaration

Implementation of three-address code – quadruples

- A record with four fields
  - *op*, *arg1*, *arg2*, *res*
- Some statements don't use an *arg* or even *res*
- Operands are references to symbol tables

	op	arg1	arg2	res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Implementation of three-address code – triples

- Avoid generating temporary variables
- A record with three fields
  - *op*, *arg1*, *arg2*
- Operands are references to the symbol tables (constants or variables) or a position of the statement that compute a value

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Implementation of three-address code – indirect triples

- One array with triples
- One array with references

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	[0]
(2)	uminus	c	
(3)	*	b	[2]
(4)	+	[1]	[3]
(5)	:=	a	[4]

	stmt
[0]	(0)
[1]	(1)
[2]	(2)
[3]	(3)
[4]	(4)
[5]	(5)

Implementation comparison

- Quadruples
  - Better for intermediate code optimization
  - They can be easily moved, names remain the same
- Triples
  - Tighter
  - Poor handling while optimizing – renumbering in the whole intermediate code
- Indirect triples
  - Good handling while optimizing – the change is only in the reference array
  - About the same memory size as quadruples

Compiler principles

Semantic analysis

Jakub Yaghob

Syntax-directed definitions

- Each grammar symbol has an associated set of attributes
  - Like a record
  - Two kinds of attributes
    - Synthesized
    - Inherited
  - Attributes can represent anything
- Attribute values defined by semantic rules assigned to grammar productions
  - The order of evaluation of semantic rules is determined by the dependency graph
  - Evaluation of semantic rules defines values of attributes

Kinds of attributes

- Each production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b=f(c_1,\ldots,c_k)$, where f is a function, c_i are grammar symbols attributes from the given production, and either
  - b is a synthesized attribute of nonterminal A
  - b is an inherited attribute of a grammar symbol on the right side of the production

Attribute grammar

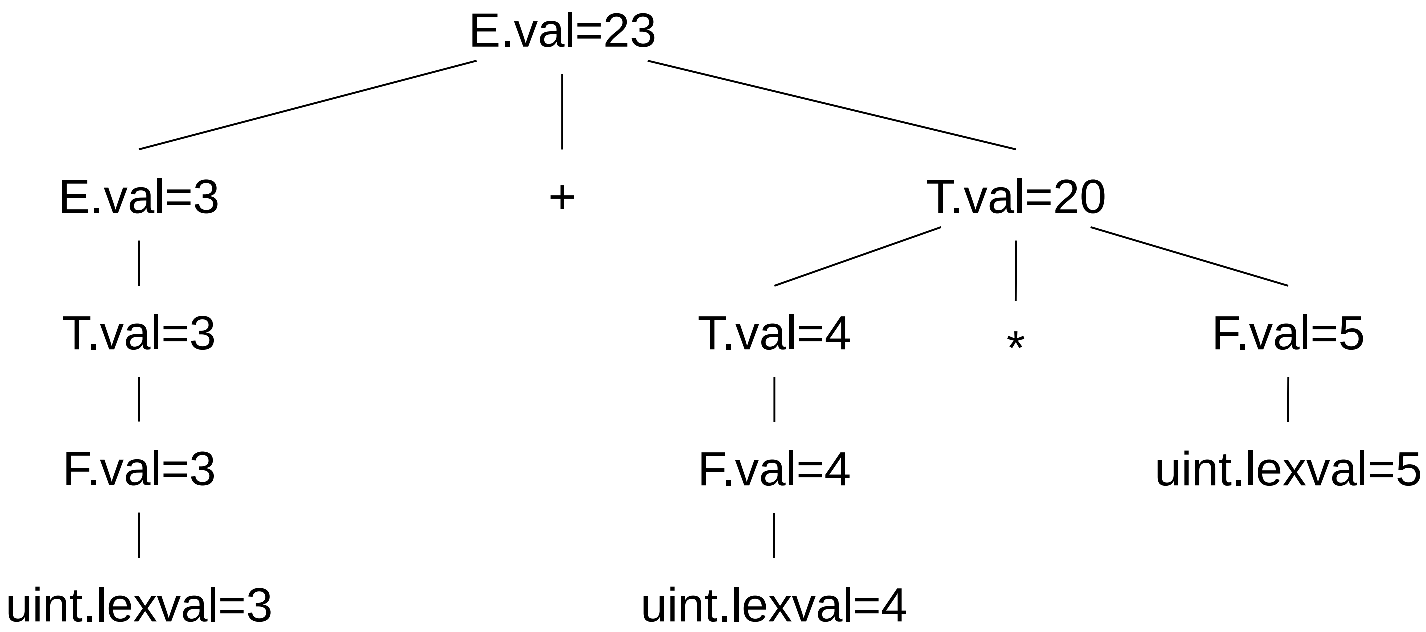
- Syntax tree with attribute values is called annotated (colored) syntax tree
- Attributed grammar is a syntax directed definition, where functions (semantic rules) don’t have side effects

Attributed grammar for our grammar

1.	$E \rightarrow E_R$ **+** T	$E.val = E_R.val + T.val$
2.	$E \rightarrow T$	$E.val = T.val$
3.	$T \rightarrow T_R$ ***** F	$T.val = T_R.val * F.val$
4.	$T \rightarrow F$	$T.val = F.val$
5.	$F \rightarrow$ **(** E **)**	$F.val = E.val$
6.	$F \rightarrow$ **uint**	$F.val = uint.lexval$

Example of annotated tree

- 3+4*5



Synthesized attributes

- An attribute of a nonterminal from the left side of a production is evaluated based on attributes of symbols from the right side of the production
- Extensively used in practice
  - E.g. expression evaluation
- Attributed grammar, which uses only synthesized attributes, is called S-attributed grammar (purely synthesized attributed grammar)
- S-attributed grammar is easily used in bottom-up analysis
  - Evaluation during reduction
  - Attributes for grammar symbols lie on parser stack

Inherited attributes

- The value of an inherited attribute is evaluated based on parent and/or siblings attributes
- They are used for expressing dependency of a syntax construction on a context
  - E.g. whether an identifier is on left or right side of an assignment
- We can always rewrite an attribute grammar to a S-attributed grammar

Example of inherited attributes

1. $D \rightarrow T\ L$ **;**	L.in = T.typ
2. $T \rightarrow$ **int**	T.typ = int
3. $T \rightarrow$ **double**	T.typ = double
4. $L \rightarrow L_R$ **, id**	L_R.in = L.in
	id.typ = L.in
5. $L \rightarrow$ **id**	id.typ = L.in

Dependency graph

- Construction for a syntax tree
  - Nodes are created for each attribute of each node of the syntax tree
  - For each semantic rule $b=f(c_1,\ldots,c_k)$ construct directed edges from a node of the dependency graph representing c_i to the node representing b

Evaluation order

- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a syntax tree can be evaluated
  - If the dependency graph contains a circle, we are not able to determine any evaluation order

L-attributed grammar

- Czech alias “jednoduše zleva-doprava 1-průchodová gramatika”
- Attributed grammar is L-attributed, if each inherited attribute of a symbol X_j on the right side of $A \rightarrow X_1 \dots X_n$ depends only on
  - The attributes of the symbols $X_1, \dots, X_{j-1}$ (to the left of X_j)
  - The inherited attributes of A
- Used for direct attribute evaluation in top-down analysis
- Every S-attributed grammar is L-attributed

Syntax tree traversal

- If the attribute grammar isn’t L-attributed grammar, it will not be possible to evaluate attributes directly during parsing
- We need to fully construct syntax tree. It will be traversed (possibly several times) during the semantic analysis. Several attributes, which we were unable to evaluate during syntax-directed translation, will be evaluated during the traversal

Static checking during translation

- Type checking
  - Incorrect operand type of an operator
    - Pointer multiplication
- Checking control flow
  - If the change in control flow is legal
    - Break statement in C, if it is not in switch or a loop
- Uniqueness checking
  - Some objects can be defined only once
    - Labels in a function, global objects identifiers
- Name checking
  - Some constructions must have the same name at the start and at the end
    - Assembler procedures

Symbolic tables

- Growing tables
  - Constants
- Stack tables
  - Identifier visibility in blocks
  - Simple implementation
    - Visibility linked list for a block (stack)
  - Used implementation
    - Identifiers „colored“ by a unique block number

Error handling

- The compiler must find all errors in the input word and must not show non-existent errors
- Error reporting
  - Clearly and accurately
  - Recover from an error quickly enough and continue in translation
  - Do not significantly slow down the processing of a correct input
- Introduced errors
  - Imprecise recovery from a previous error causes inception of non-existent errors

Types of errors

- Lexical errors
  - Malformed lexical elements
    - Unfinished string and the EOL
  - Error recovery by ignoring the error
- Syntax errors
  - The input word is not in an input language
    - Unpaired parenthesis
- Semantic errors
  - Static checks
    - Undeclared variable, wrong number of parameters in a function call, wrong type used with an operator
- Logical errors
  - Errors in programming
    - Indefinite loop, uninitialized variable

Syntax errors recovery

- Panic mode
  - A set of skeletal symbols
  - When an error is encountered, skip all symbols until a symbol from the skeletal set is found
  - Then the parser is put into a known state
- Productions modifications
  - Insert, remove, replace a terminal in a production
- Intentional error production
  - Grammar augmentation with usual errors with specific error message
    - E.g. assignment in Pascal

Compiler principles

Intermediate code generation

Jakub Yaghob

Syntax-directed translation into three-address code

- Add several attributes to each grammar symbol
  - Placement – a name of an object holding a value of the object
  - Code – a sequence of three-address instructions evaluating the symbol
  - Label – an absolute or relative address to three-address code

Example for our grammar

- $E \rightarrow E_R + T$

  $E.p = newtemp$

  $E.c = E_R.c \mid T.c \mid gen(E.p = E_R.p + T.p)$

- $E \rightarrow T$

  $E.p = T.p$

  $E.c = T.c$

- $T \rightarrow T_R * F$

  $T.p = newtemp$

  $T.c = T_R.c \mid F.c \mid gen(T.p = T_R.p * F.p)$

- $T \rightarrow F$

  $T.p = F.p$

  $T.c = F.c$

- $F \rightarrow (E)$

  $F.p = E.p$

  $F.c = E.c$

- $F \rightarrow id$

  $F.p = id.p$

  F.c = ‘’

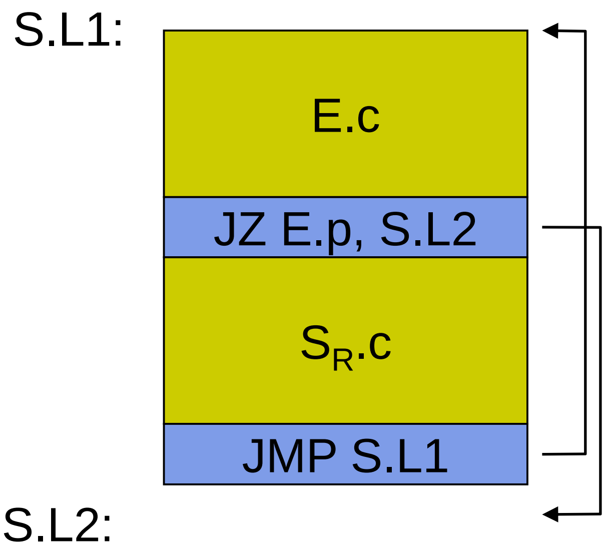
Example for while (amateur)

- S→ **while** E **do** S_R

S.L1 = curradr

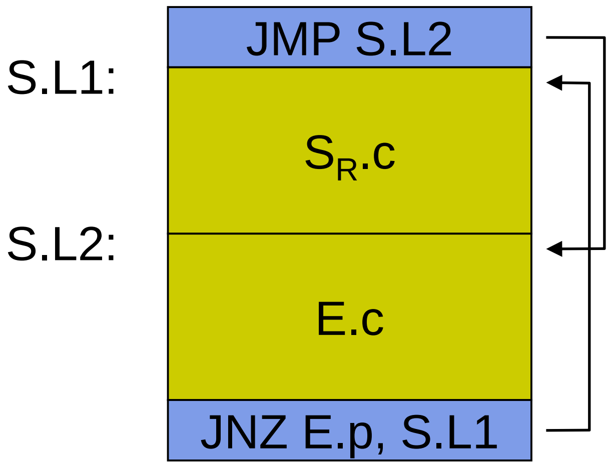
S.L2 = curradr + E.c.size + S_R.c.size + 2

S.c = E.c | gen(JZ E.p, S.L2) | S_R.c | gen(JMP S.L1)



Example for while (professional)

- $S \rightarrow$ **while** E **do** S_R



$S.L1 = curradr + 1$

$S.L2 = curradr + S_R.c.size + 1$

$S.c = gen(JMP\ S.L2) \mid S_R.c \mid E.c \mid gen(JNZ\ E.p,\ S.L1)$

Declarations

- Global objects declarations
  - E.g. global variables in C
- Local objects declarations
  - E.g. local variables in a function
  - At the start of the function × at the start of a block × in the block body
- Moving declarations to a “good” place
  - Global declarations in one well-defined place
  - Local declarations at the start of a function
    - Live-variable in the function
- Determine the size of the object
- Determine the offset of fields in a structure
  - It does not have to be solved in intermediate code

Assignment

- Type conversion
  - Usually explicit
    - Intermediate code does not know input language semantic
- Structure field access
  - Use calculated offset from declaration
  - Direct access using a pointer and added constant
- Array unfolding
  - Multidimensional-array are perceived as one-dimensional (like a memory) – stored in one of the two forms, either row-major (row-by-row) or column-major (column-by-column)
  - One-dimensional array A[lb..ub] of type with width w
    - base + (i-lb) * w
    - i * w + (base – lb * w)
  - Two-dimensional array $A[lb_1..ub_1,lb_2..ub_2]$
    - $base + ((i_1-lb_1) * (ub_2-lb_2+1) + i_2-lb_2) * w$

Boolean expression

- Sometimes replaced FALSE=0, TRUE=1 (or anything !=0)
- Numeric (full) evaluation
  - All parts of the expression are evaluated
  - Pascal
- Short evaluation
  - if the result is already known during the evaluation, it won’t be evaluated further
    - Jumps in the evaluation
  - C

Switch

- What we need
  - Evaluate the expression
  - Find which value in the list of cases is the same as the value of the expression
  - Execute the statement associated with the value found
- Finding the case
  - Sequence of conditional branches
    - Small number of cases (<10)
  - Binary search in table [value,caseptr]
  - Binary tree of conditions
  - Table of pointers indexed by value
    - High density of values in the range of case values

Backpatching

- Single pass translation: how to set a destination address in a jump to a forward label?
  - Forward jump
  - Calling a not yet defined function in a module
    - Can be resolved by a linker
- Generate branches with the targets of the jumps temporarily left unspecified
- For each label remember a set of instructions referencing it
- Determine the address of the label
- Go through the set of instructions and *backpatch* the determined address

Procedure calls

- Evaluate parameters
- Pass parameters by value or by reference
  - VAR or non-VAR parameters in Pascal
- Binding real and formal parameters
  - Positional
  - By name
- Return value

Compiler principles

High-level optimization

Jakub Yaghob

Optimization

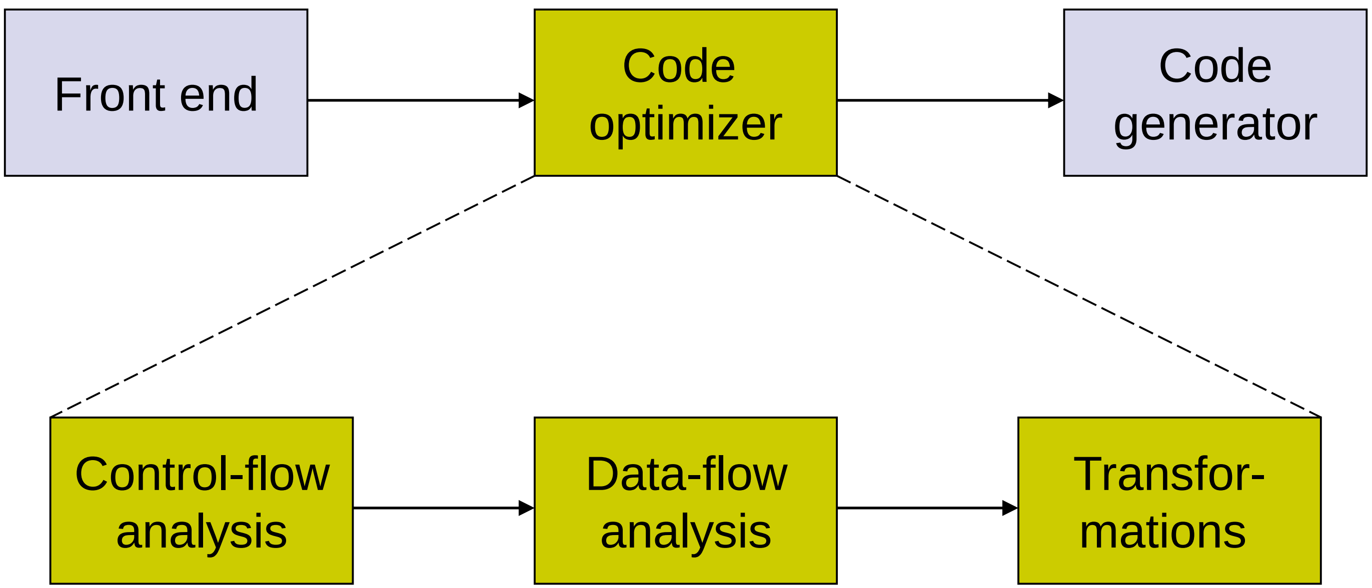
- Ideal situation – generated code is equal to manually written code
  - Is it valid today?
- Reality
  - Only in several defined situations
  - It is difficult
- Optimization
  - Program transformation
  - For speed or size
- High-level optimization
  - Intermediate code

Criteria for transformation

- Preserve the meaning of programs
  - Output and program behavior must not be changed by a transformation – always use safe approach
- Speed up programs by a measurable amount
  - Sometimes space reduction
  - Occasionally the optimization can slow down a program, on the average it improves it
- A transformation must be worth the effort
  - Complex/slow optimization with negligible effect are not worth the effort

Organization of the code optimizer

- Control-flow analysis
  - Construct the control flow graph
- Data-flow analysis
  - Live variables



Control flow graph

- Three-address code graph representation
- Nodes represent computation
- Directed edges represent control flow
- Important for optimization and representation

Basic block

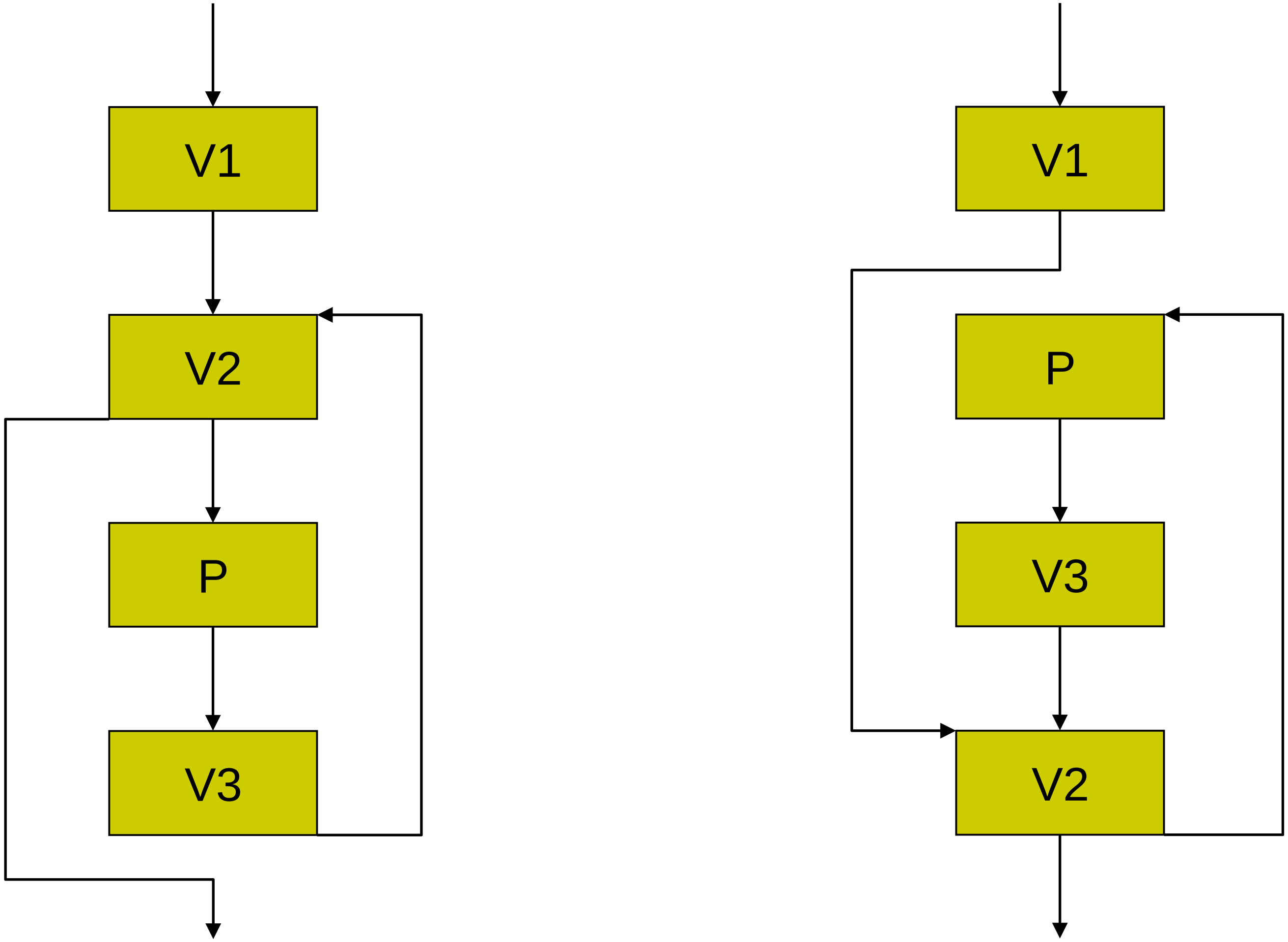
- A sequence of consecutive three-address statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
- Algorithm for partitioning a sequence of three-address statements into basic blocks
  - Determine the set of leaders
    - The first statement is a leader
    - Any statement that is the target of a jump is a leader
    - Any statement that immediately follows a jump is a leader
  - For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the sequence

Construction of control flow graph

- A directed graph
- The nodes are the basic blocks
- One node is initial – a block whose leader is the first statement in the sequence
- There is a directed edge from block B_1 to block B_2, if one of the following condition holds
  - The last statement of B_1 is a jump to the leader of B_2
  - B_2 immediately follows B_1 and B_1 does not end in an unconditional jump

Example of control flow graph

- for(V1;V2;V3) P



Data-flow analysis

- Assign live variables to nodes of the control-flow graph
  - Variables need not to be live in the whole basic block
  - Compute exact lifespan inside the basic block
- Evaluation order
  - Topological sort of DAG
    - Nodes are definitions and uses of a variable
    - Oriented edges are from a use to a definition in a computation
- Pointer modeling
  - Pointer aliasing
- Function calls modeling
  - Interprocedural optimization

Live variable analysis

- Three-address instruction x := y op z *defines* x and *uses* y and z
- A variable is *alive* at some point of the intermediate code, if the point lies on a path from the point, where it is defined, to a point, where it is used, at the same time there are no other definitions on the path

Kinds of optimizations

- Inside a basic block
  - No jumps
- Inside a function
  - Code movement between basic blocks
  - Create, remove basic blocks
- Whole program (interprocedural)
  - Speed up calls between functions
  - During linker phase with delayed compilation

Local optimization

- Inside a one basic block
  - Common subexpression elimination (CSE)
  - Copy propagation
  - Dead-code elimination
    - x = y+z, and *x* is not used any more
  - Constant folding
  - Algebraic transformations/identities
    - x = x + 0, x = x * 1

Local CSE

- a = b + c
- b = a – d
- c = b + c
- d = a – d

- a = b + c
- b = a – d
- c = b + c
- d = b

- There is a hidden problem
  - Pointer aliasing
  - ap = &a
  - b = a + c
  - *ap = d
  - e = a + c

Global optimization

- Among more basic blocks, possible movement of a code between BB, control-flow graph modification
  - Common subexpression elimination
  - Copy propagation
  - Dead-code elimination
  - Loop optimization
    - Invariant code motion
    - Reduction in strength of operation
    - Removing induction variable

Loop optimization – 1

- Invariant code motion
  - Expression yields the same result independent of the number of times a loop is executed

```
while(i<limit-5) S
```

```
t = limit – 5;
while(i<t) S
```

Loop optimization – 2

- Reduction in strength of operation
  - Transform multiplication to addition
- Removing induction variable
  - Only one induction variable for one loop
  - Usually removed during reduction in strength

```
for(i=3;i<8;i+=3)
{
  j = i*2;
  a[j] = j;
}
```

```
for(t=6;t<16;t+=6)
  a[t] = t;
```

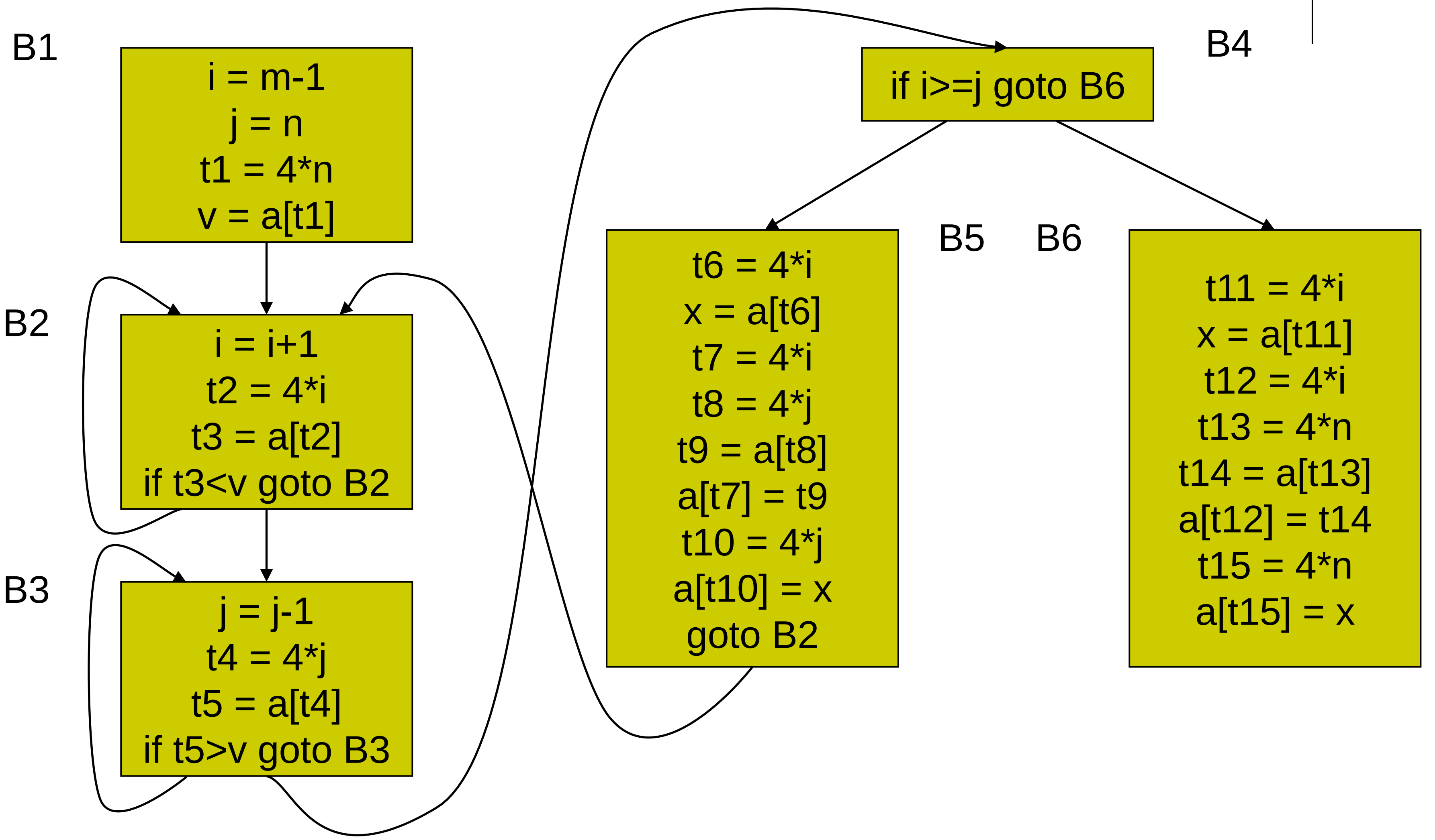
Example – C code

```
i=m-1; j=n; v=a[n];
for(;;) {
  do ++i; while(a[i]<v);
  do --j; while(a[j]>v);
  if(i>=j) break;
  x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

Example – three-address code

1) i = m-1
2) j = n
3) t1 = 4*n
4) v = a[t1]
5) i = i+1
6) t2 = 4*i
7) t3 = a[t2]
8) if t3<v goto 5
9) j = j-1
10) t4 = 4*j
11) t5 = a[t4]
12) if t5>v goto 9
13) if i>=j goto 23
14) t6 = 4*i
15) x = a[t6]
16) t7 = 4*i
17) t8 = 4*j
18) t9 = a[t8]
19) a[t7] = t9
20) t10 = 4*j
21) a[t10] = x
22) goto 5
23) t11 = 4*i
24) x = a[t11]
25) t12 = 4*i
26) t13 = 4*n
27) t14 = a[t13]
28) a[t12] = t14
29) t15 = 4*n
30) a[t15] = x

Example – control-flow graph



Example - LCSE

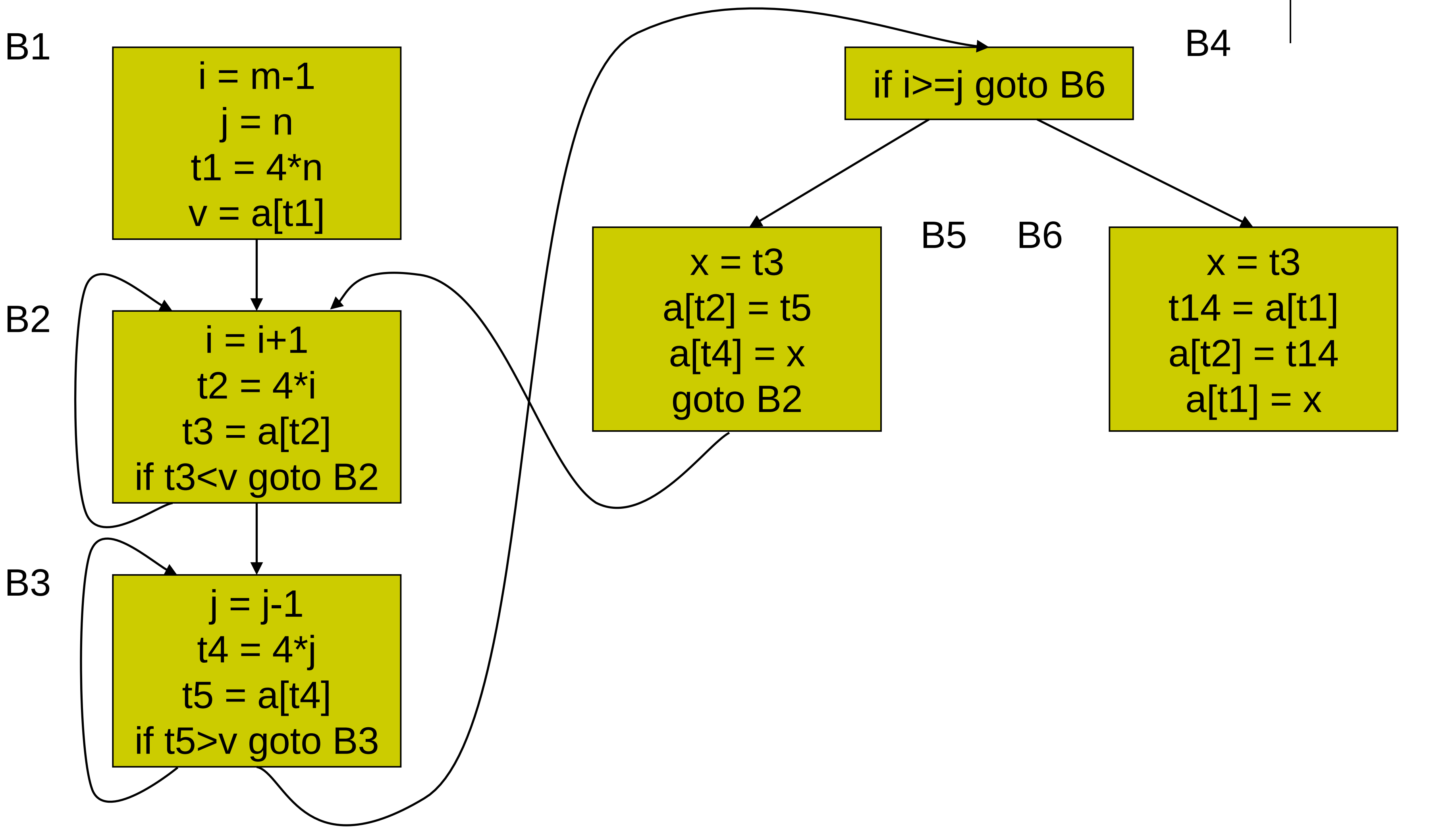
B5

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B5

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

Example – GCSE



Example – copy propagation and dead-code elimination

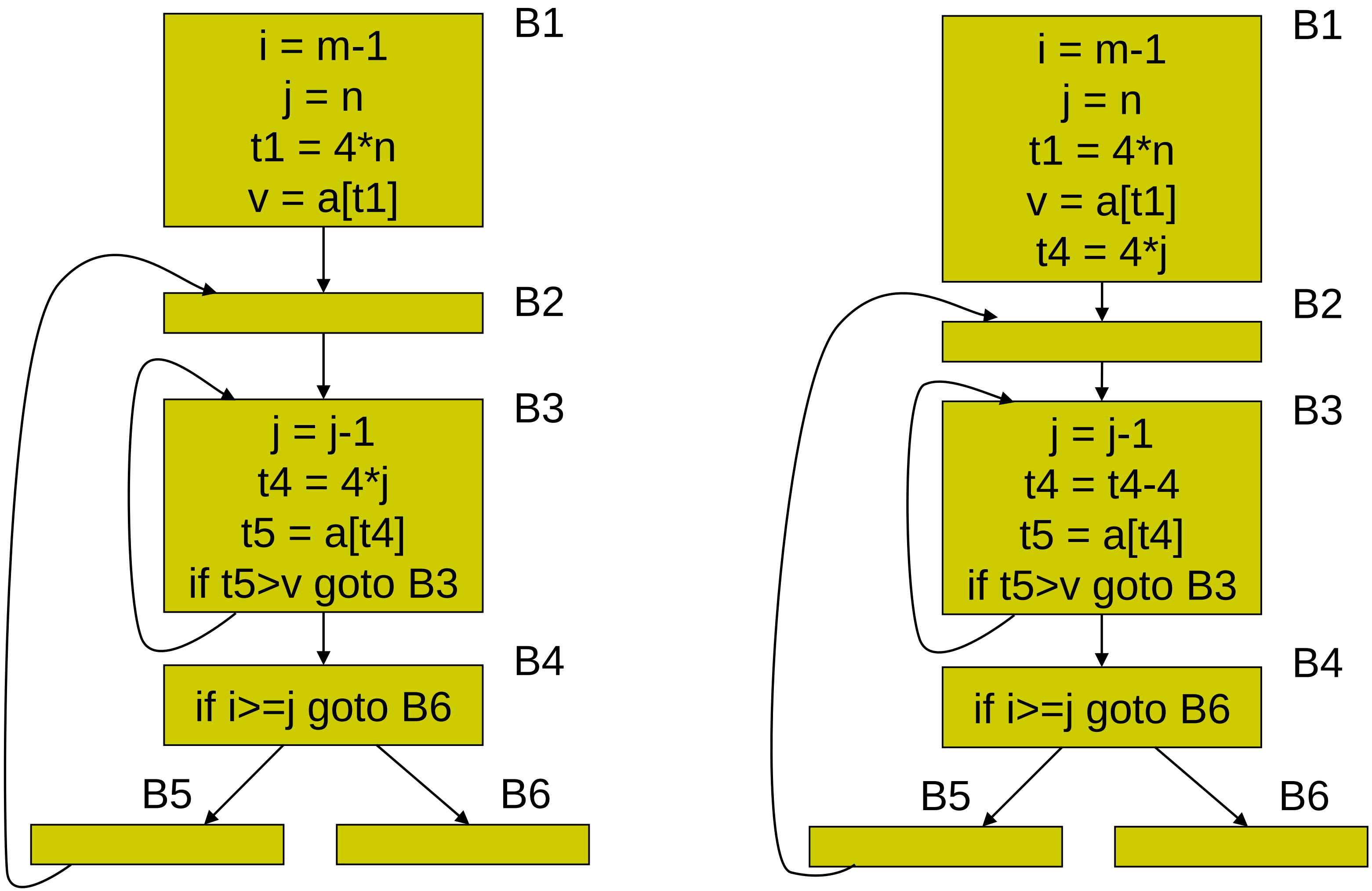
B5
```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

B5
```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

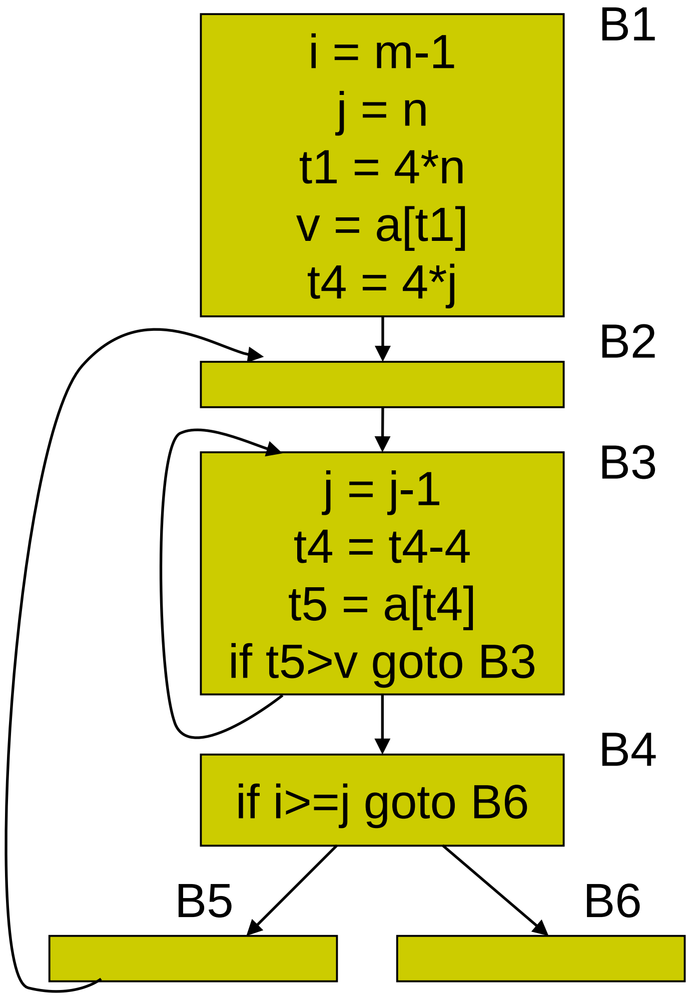
B5
```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

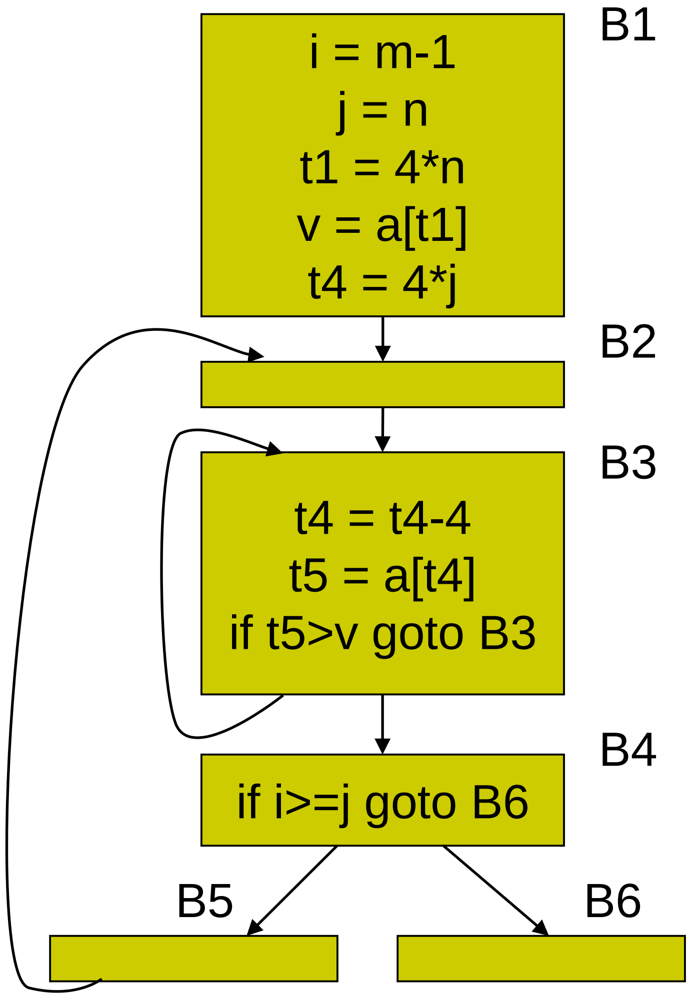
B5
```
a[t2] = t5
a[t4] = t3
goto B2
```

Example – reduction in strength

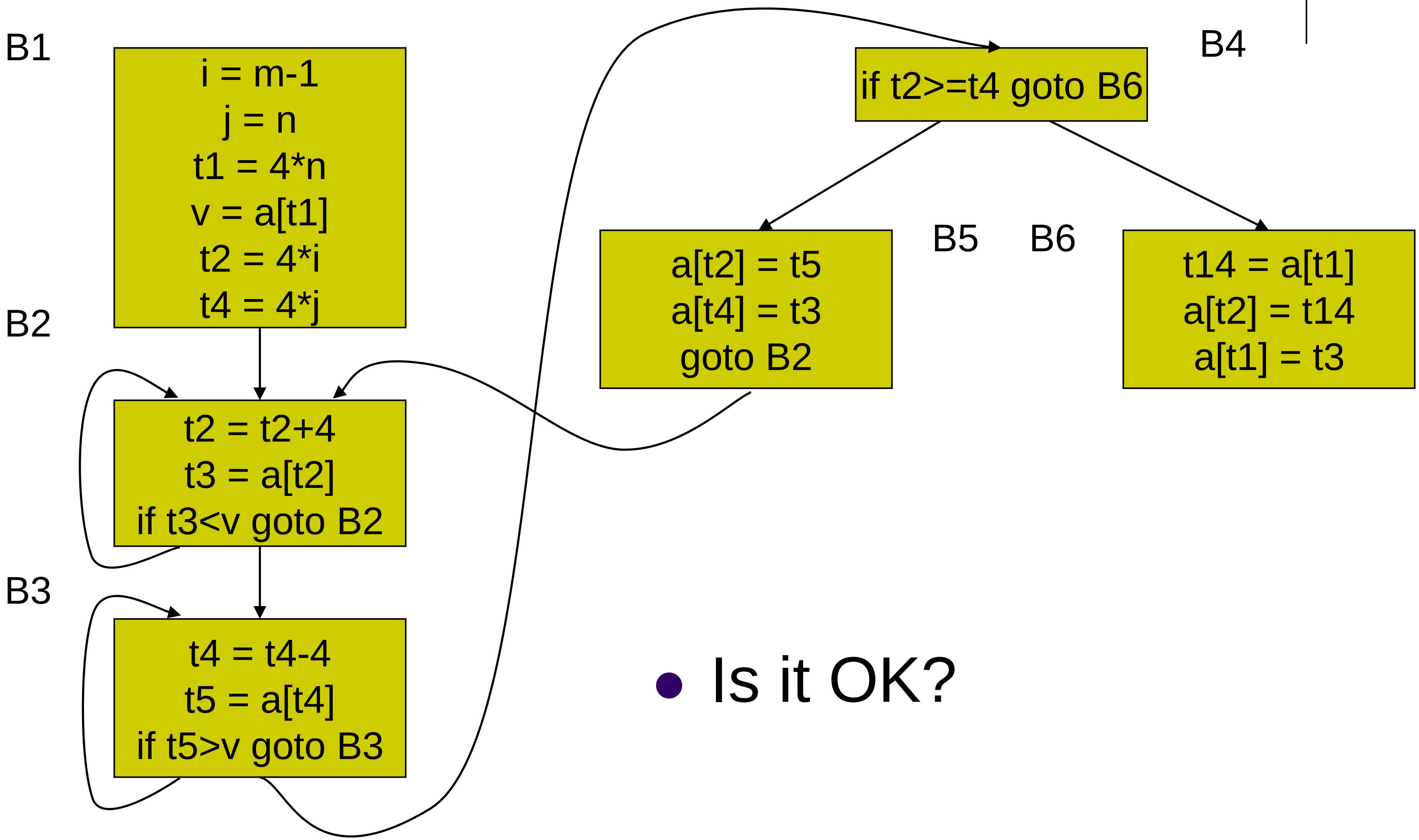


Example – removing induction variable



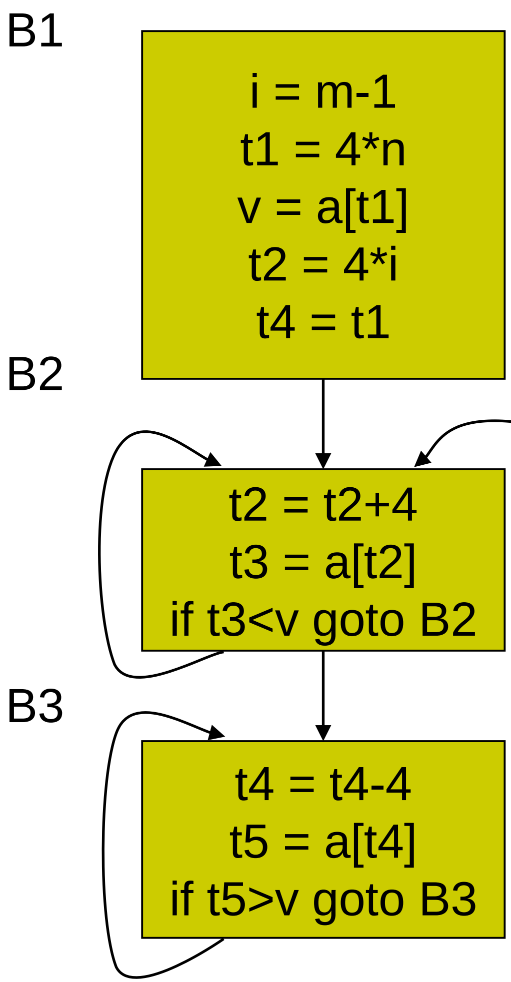


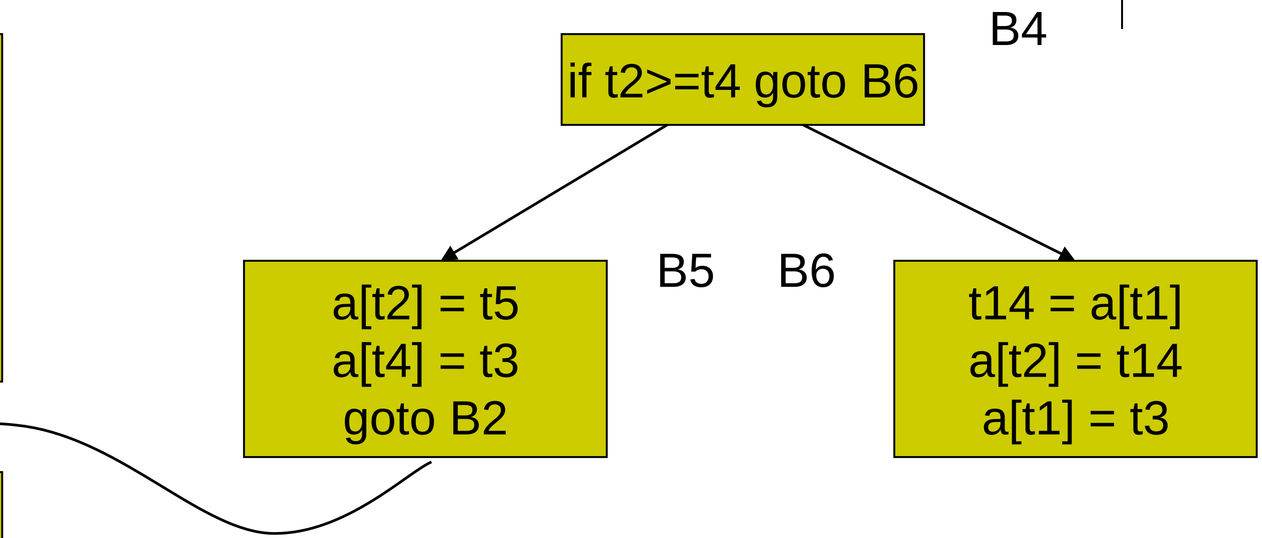
Example – result



- Is it OK?

Example – final result





- Another application of LCSE on B1

Parallelization and vectorization

- Parallelization
  - Implicit × explicit
  - Code parts allowing concurrent execution
  - Prefetch variables to a cache
- Vectorization
  - Expression parallelization using SIMD operations

Profile Guided Optimization

- Three phases
  - Instrumentation
    - Specially compiled code with calls to “collecting” functions at the start and the end of each basic block
  - Profiling
    - Run the instrumented code with a “typical” input
    - Collect statistical data about visiting collecting functions
  - Optimization
    - Collected data influence optimization
    - Used in optimization upon intermediate code and code
    - Adding weights to edges in control-flow graph
    - Looking for most important paths in control-flow graph
    - Register assignment

Compiler principles

Processor architectures

Jakub Yaghob

Processor architectures

- Processor architecture is a target language
  - For compilers targeting a processor code
  - It influences the compiler backend, namely code generator
  - It can affect an intermediate code form and instructions, used high-level optimizations

Registers

- The fastest memory
- Precious resource
  - x86 has 7 of 32-bits integer registers
  - IA-64 has 128 of 64-bits integer registers
- Different register types for different data types
  - Integer, floating point, address, vector
- Different access modes
  - Direct
  - Stack (FPU)

Instruction set

- RISC
  - Simple instructions, small number (e.g. no division)
- CISC
  - Complex instructions, wide repertoire
- Load-Execute-Store
- Orthogonality
  - x86 has non-orthogonal instruction set

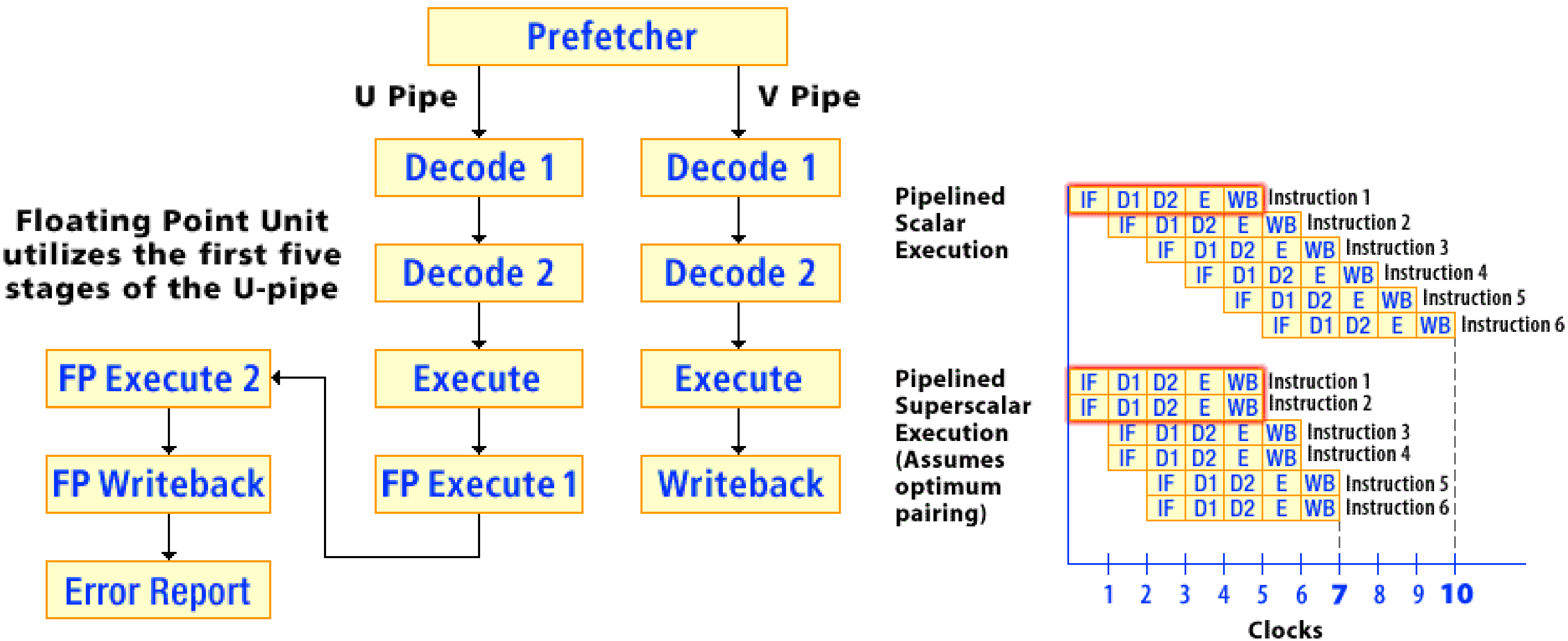
Pipelining

- Next instruction fetch and decode started before the previous instruction was finished
- Each stage and each instruction has a latency
  - Problem with operand dependency (RAW)

IF	ID	EX	MEM	WB				
i	IF	ID	EX	MEM	WB			
t		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Superscalar processor

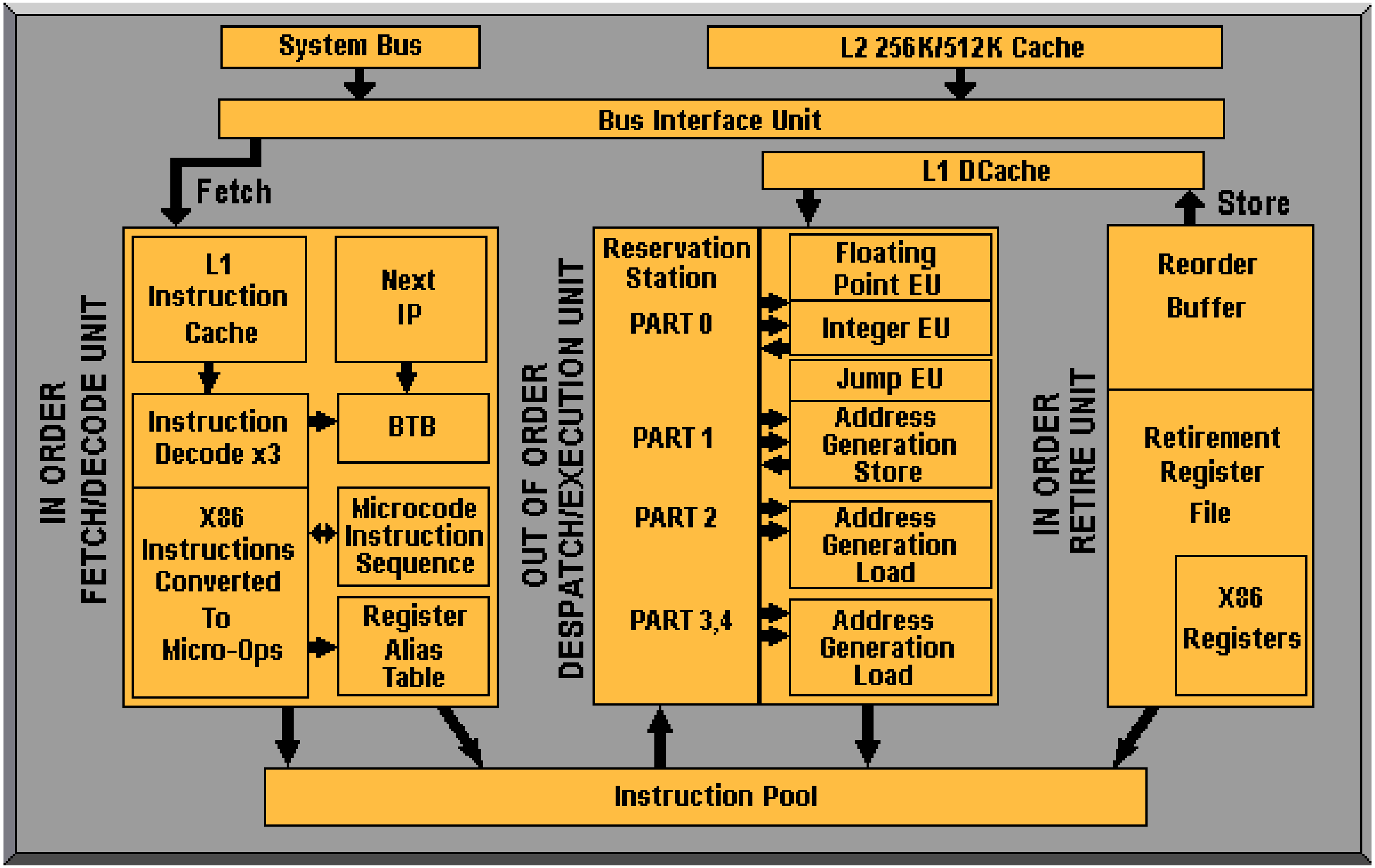
- More equal units capable of parallel instruction execution



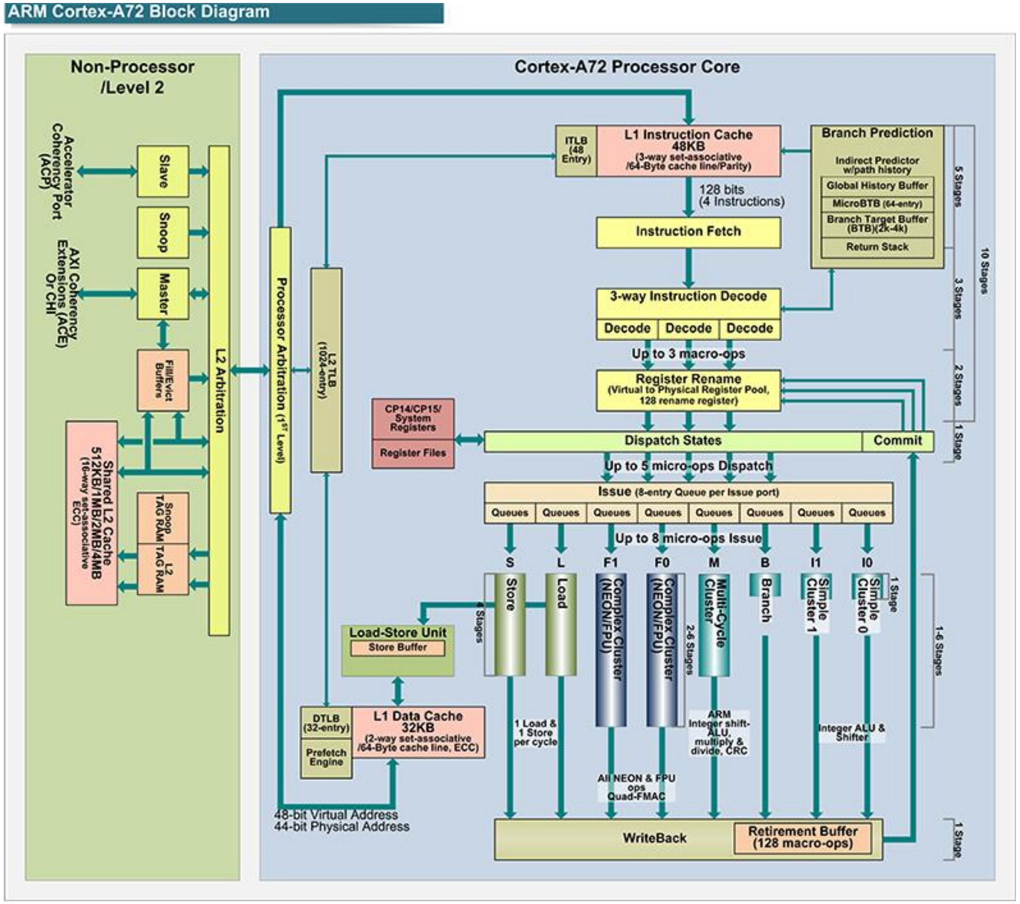
Out-of-Order Execution

- Instruction fetch
- Instruction dispatch to an instruction queue
- The instruction waits until its input operands are available
- The instruction is issued to its execution unit
- The results are queued
- When all older instructions written their results to the register file, the instruction writes its result

Out-of-Order Execution – AMD



Out-of-Order Execution – ARM Cortex-A72



Branch prediction

- Deep pipelines have a problem with not taken conditional jumps
- Dynamic branch prediction
  - BTB
  - CPU uses history (patterns of some length) for branch prediction
  - Static prediction used when no history is available
- Static branch prediction
  - No hint
    - Forward jump is not taken, backward jump is taken
  - Hint
    - A compiler computes branch probability and generates appropriate branch hint
- Delay slot

Speculative execution

- Execution of a code in advance which is not used later
- Significant disproportion between CPU and memory speed
- Usually used for read operations in advance
- CPU postpones write operations
- Memory barriers
- Code speculation
  - Check for successful execution
- Data speculation
  - Moved forward even with possible pointer-aliasing

SIMD instructions

- Sometimes called multimedia instructions
- Integer and float data types
- Expression vectorization already in intermediate code or later in code generation
- Considerable (constant) execution speedup
- Sometimes difficult detection of vectorization opportunity

VLIW

- Instruction encoding
  - Templates in IA-64
- ILP (Instruction Level Parallelism)
  - Concurrency encoded directly in instructions
  - Dependencies in concurrency group
    - RAW, WAW disallowed
    - WAR allowed

SoC (System on Chip) + microcontrollers

- Small memory space
  - Optimization for size of code and data
    - One-bit variables
- Separated memory spaces
  - Code and several data address spaces
  - Different access
- Simple pipeline, no superscalarity

Code generation

- Memory allocation
- Instruction selection
- Register allocation
- Instruction scheduling
- Output
  - Output file format
  - Objects
    - Code, data, relocations, external and public symbols, debug information

Memory allocation

- Code
  - Jump resolving
- Static data
  - Global data
- Stack
  - Local variables and function parameters
- Size of data types
- Memory placement
- Data alignment
- Decide, what is in memory and what gets an allocated register

Instruction selection

- 1:N
  - One intermediate code instruction corresponds with N target instructions
  - Simple, the generated code is considerably suboptimal
- M:N
  - M intermediate code instructions correspond with N target instructions
  - NP-complete problem of selection
  - Heuristics
- More possibilities of code generation
  - CISC, SIMD, VLIW
- Non-orthogonal instruction set problem

Register allocation

- What is placed to the memory and what is placed into a register of a required register type
- Non-orthogonal instruction set problem
- Coloring of a graph of variable liveness
  - The graph is constructed only for a selected variable type (e.g. integer variables)
  - The number of colors is equal to the number of available registers of the selected type
  - When there is no graph coloring, remove “less important” graph nodes and try again
  - We don’t need to find an optimal graph coloring, we are looking for any graph coloring

Instruction scheduling

- Rearrange instructions keeping semantic and avoiding pipeline stalls
  - RAW, WAW, WAR dependencies
  - Construct graph of data dependency
    - Oriented graph, where an edge represents instruction ordering for data availability
    - Find any topological order
  - Instruction latency, speculations, superscalarity, out-of-order execution
  - Usually performed upon one basic block

Compiler principles

Processor architectures

Jakub Yaghob

Processor architectures

- Processor architecture is a target language
  - For compilers targeting a processor code
  - It influences the compiler backend, namely code generator
  - It can affect an intermediate code form and instructions, used high-level optimizations

Registers

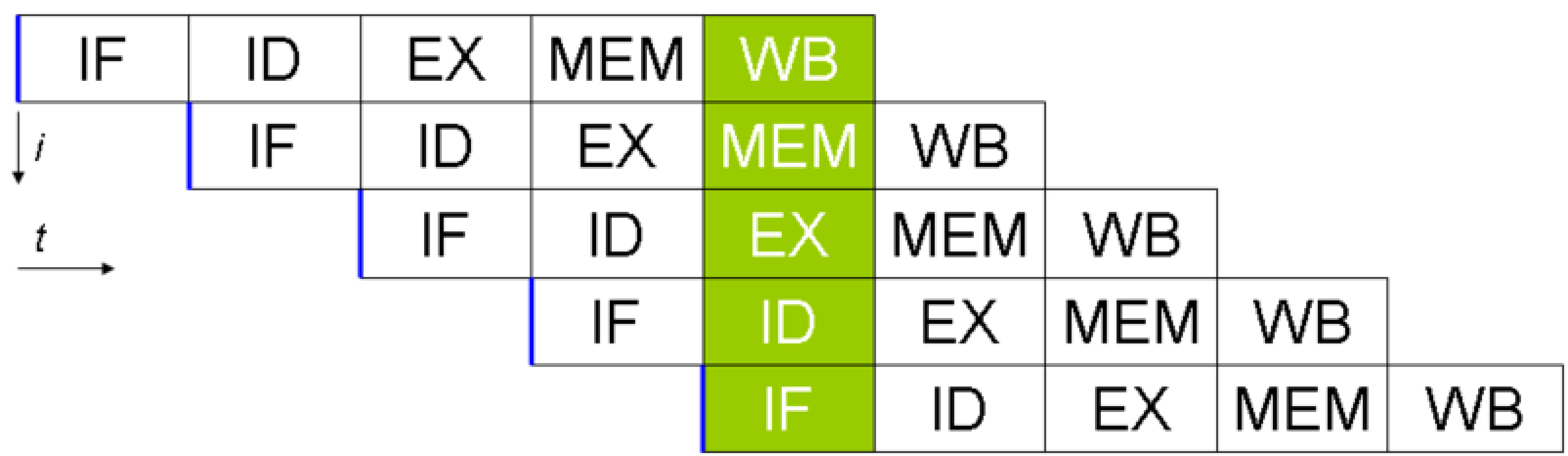
- The fastest memory
- Precious resource
  - x86 has 7 of 32-bits integer registers
  - IA-64 has 128 of 64-bits integer registers
- Different register types for different data types
  - Integer, floating point, address, vector
- Different access modes
  - Direct
  - Stack (FPU)

Instruction set

- RISC
  - Simple instructions, small number (e.g. no division)
- CISC
  - Complex instructions, wide repertoire
- Load-Execute-Store
- Orthogonality
  - x86 has non-orthogonal instruction set

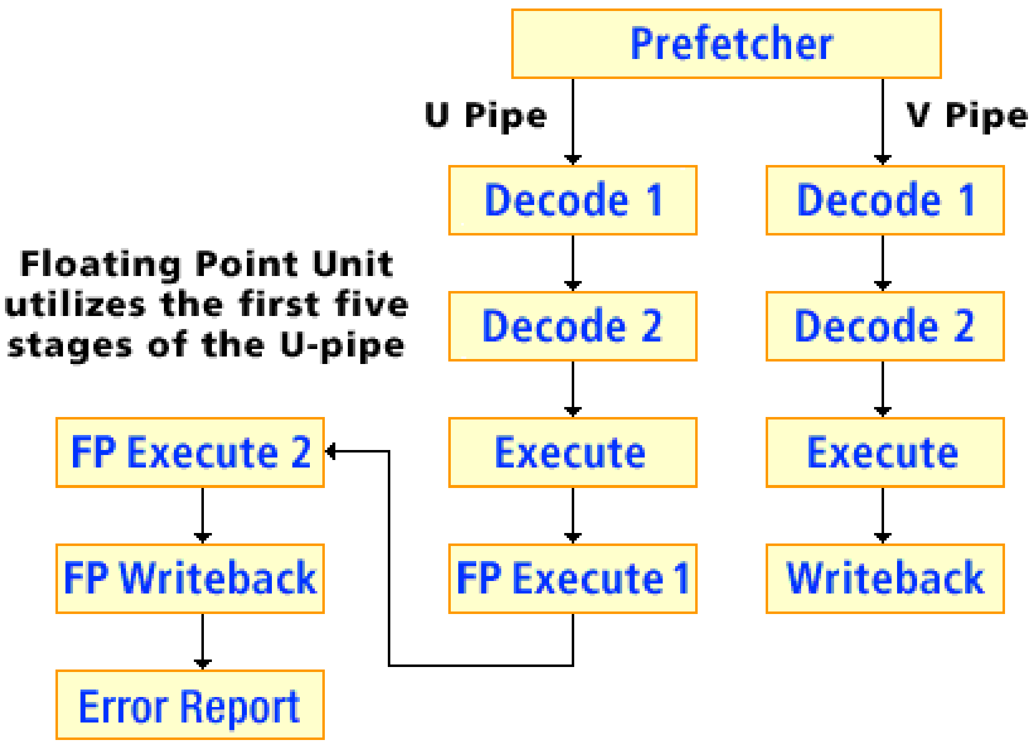
Pipelining

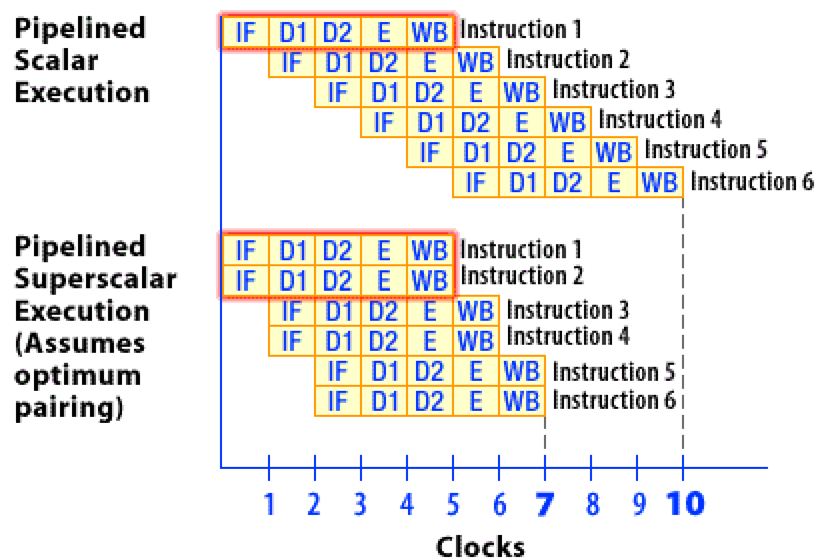
- Next instruction fetch and decode started before the previous instruction was finished
- Each stage and each instruction has a latency
  - Problem with operand dependency (RAW)



Superscalar processor

- More equal units capable of parallel instruction execution

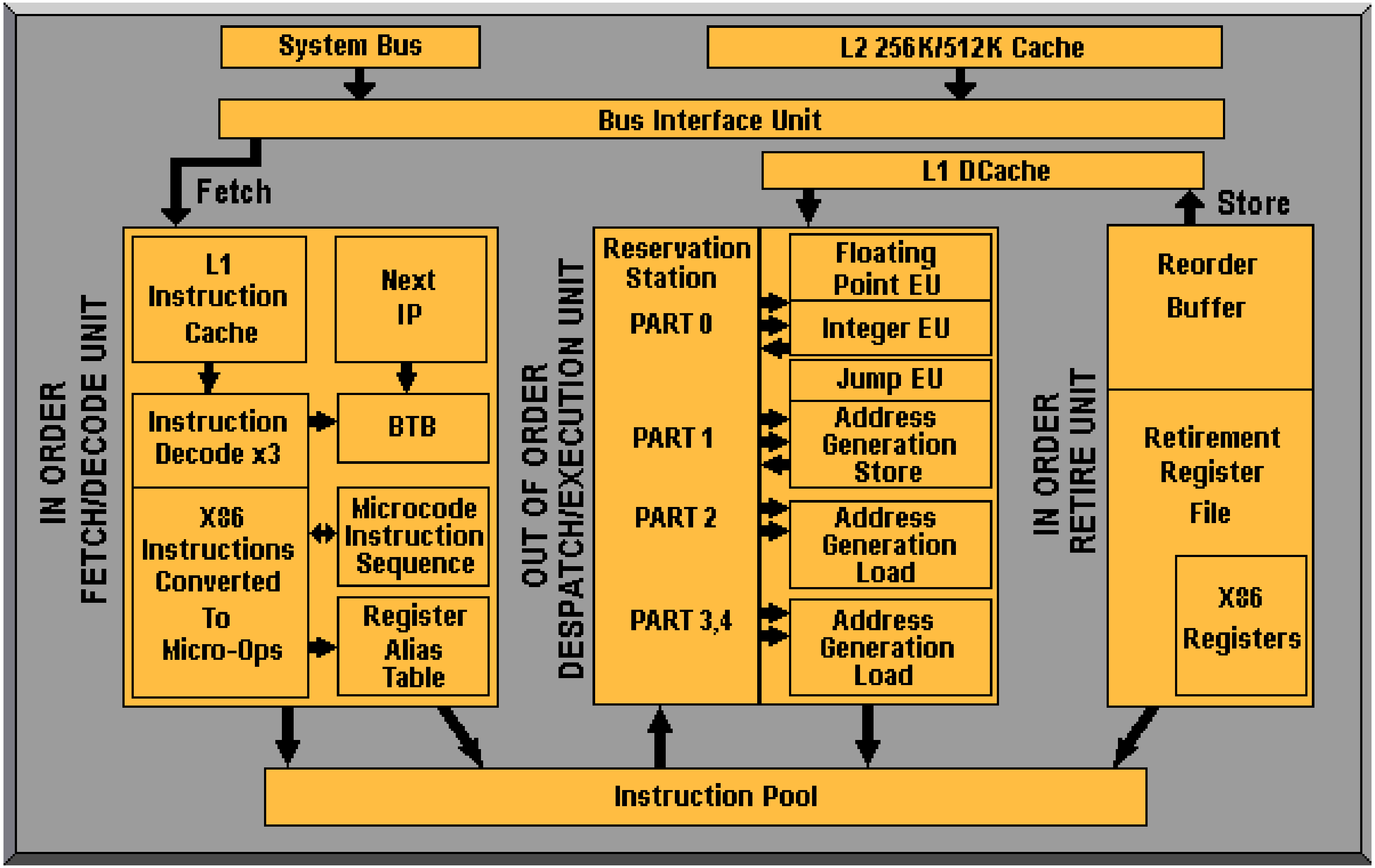




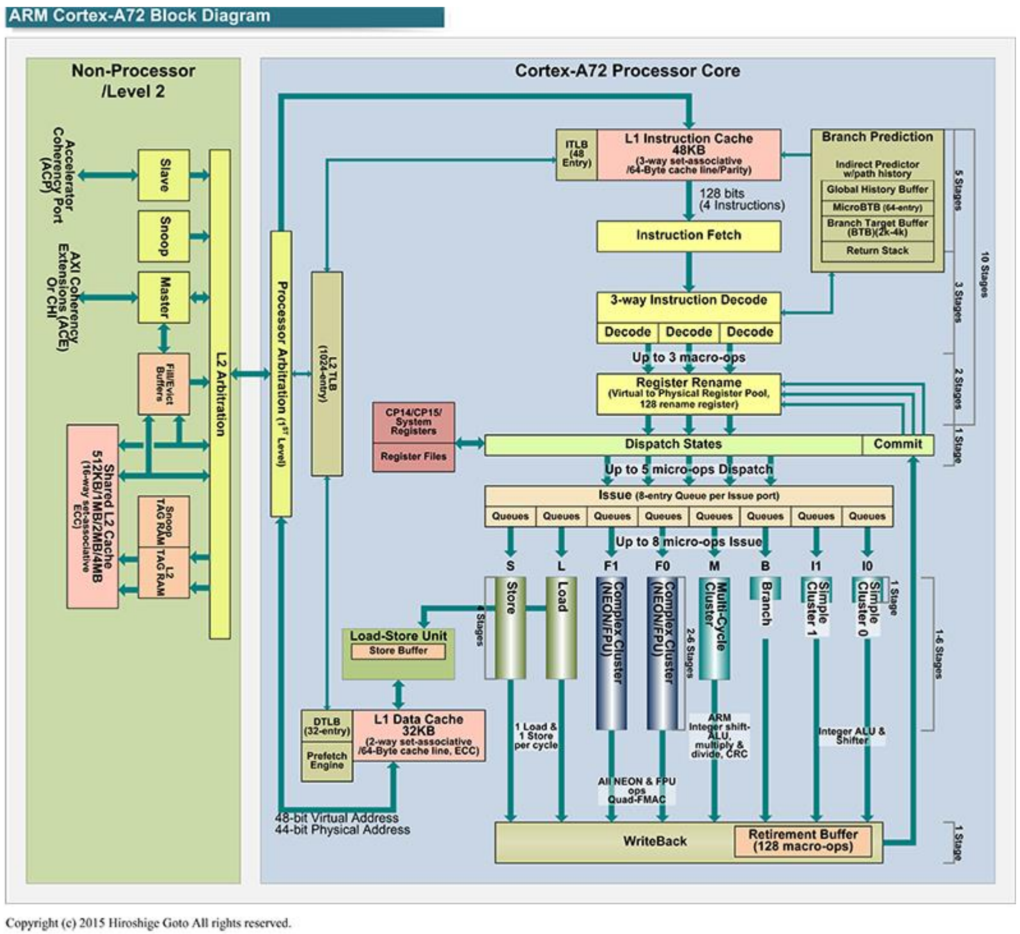
Out-of-Order Execution

- Instruction fetch
- Instruction dispatch to an instruction queue
- The instruction waits until its input operands are available
- The instruction is issued to its execution unit
- The results are queued
- When all older instructions written their results to the register file, the instruction writes its result

Out-of-Order Execution – AMD



Out-of-Order Execution – ARM Cortex-A72



Branch prediction

- Deep pipelines have a problem with not taken conditional jumps
- Dynamic branch prediction
  - BTB
  - CPU uses history (patterns of some length) for branch prediction
  - Static prediction used when no history is available
- Static branch prediction
  - No hint
    - Forward jump is not taken, backward jump is taken
  - Hint
    - A compiler computes branch probability and generates appropriate branch hint
- Delay slot

Speculative execution

- Execution of a code in advance which is not used later
- Significant disproportion between CPU and memory speed
- Usually used for read operations in advance
- CPU postpones write operations
- Memory barriers
- Code speculation
  - Check for successful execution
- Data speculation
  - Moved forward even with possible pointer-aliasing

SIMD instructions

- Sometimes called multimedia instructions
- Integer and float data types
- Expression vectorization already in intermediate code or later in code generation
- Considerable (constant) execution speedup
- Sometimes difficult detection of vectorization opportunity

VLIW

- Instruction encoding
  - Templates in IA-64
- ILP (Instruction Level Parallelism)
  - Concurrency encoded directly in instructions
  - Dependencies in concurrency group
    - RAW, WAW disallowed
    - WAR allowed

SoC (System on Chip) + microcontrollers

- Small memory space
  - Optimization for size of code and data
    - One-bit variables
- Separated memory spaces
  - Code and several data address spaces
  - Different access
- Simple pipeline, no superscalarity

Code generation

- Memory allocation
- Instruction selection
- Register allocation
- Instruction scheduling
- Output
  - Output file format
  - Objects
    - Code, data, relocations, external and public symbols, debug information

Memory allocation

- Code
  - Jump resolving
- Static data
  - Global data
- Stack
  - Local variables and function parameters
- Size of data types
- Memory placement
- Data alignment
- Decide, what is in memory and what gets an allocated register

Instruction selection

- 1:N
  - One intermediate code instruction corresponds with N target instructions
  - Simple, the generated code is considerably suboptimal
- M:N
  - M intermediate code instructions correspond with N target instructions
  - NP-complete problem of selection
  - Heuristics
- More possibilities of code generation
  - CISC, SIMD, VLIW
- Non-orthogonal instruction set problem

Register allocation

- What is placed to the memory and what is placed into a register of a required register type
- Non-orthogonal instruction set problem
- Coloring of a graph of variable liveness
  - The graph is constructed only for a selected variable type (e.g. integer variables)
  - The number of colors is equal to the number of available registers of the selected type
  - When there is no graph coloring, remove “less important” graph nodes and try again
  - We don’t need to find an optimal graph coloring, we are looking for any graph coloring

Instruction scheduling

- Rearrange instructions keeping semantic and avoiding pipeline stalls
  - RAW, WAW, WAR dependencies
  - Construct graph of data dependency
    - Oriented graph, where an edge represents instruction ordering for data availability
    - Find any topological order
  - Instruction latency, speculations, superscalarity, out-of-order execution
  - Usually performed upon one basic block

Compiler principles

Run-time support

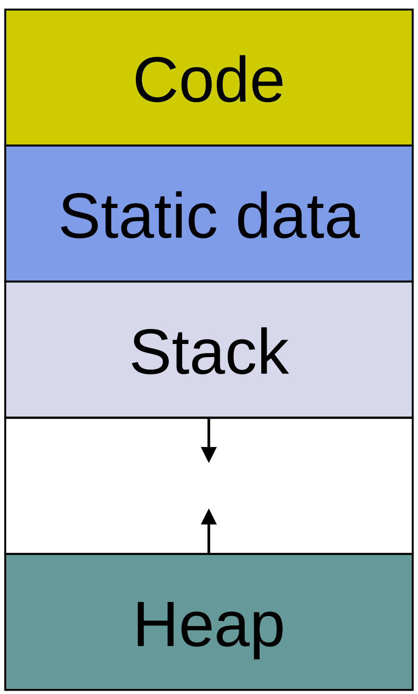
Jakub Yaghob

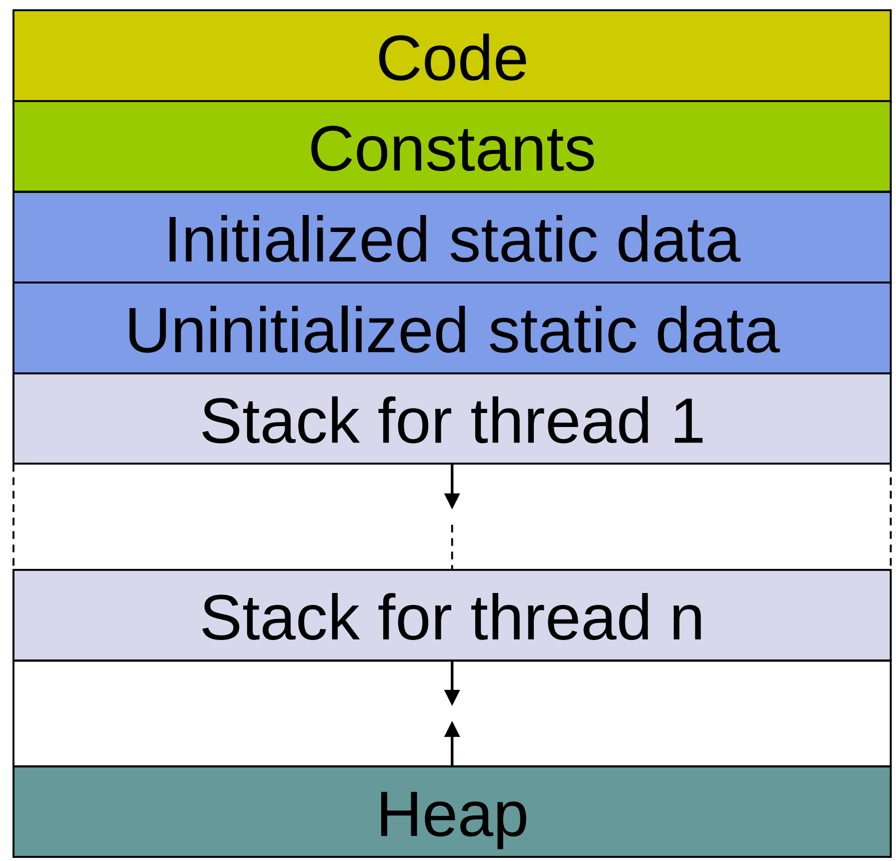
Run-time support

- Static language support
  - Compiler
  - Library interface
    - Header files
- Dynamic language support
  - Run-time program environment
    - Storage organization
    - Memory content before execution
    - Constructors and destructors of global objects
  - Libraries
  - Calling convention

Memory organization

- Memory organization during procedural program execution





Activation record

Return value
Actual parameters
Control link
Access link
Saved machine status
Local data
Temporaries

- Control link
  - Activation record of the caller
- Access link
  - Pointer to nonlocal data held in other activation records
- Saved machine status
  - Return address to the code
  - Registers

Calling convention

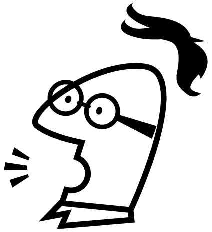
- Calling convention
  - Public name mangling
  - Call/return sequence for functions and procedures
    - Housekeeping responsibility
  - Parameter passing
    - Registers, stack
    - Order of passed parameters
  - Return value
    - Registers, stacks
  - Registers role
    - Parameter passing, scratch, preserved

Public name mangling

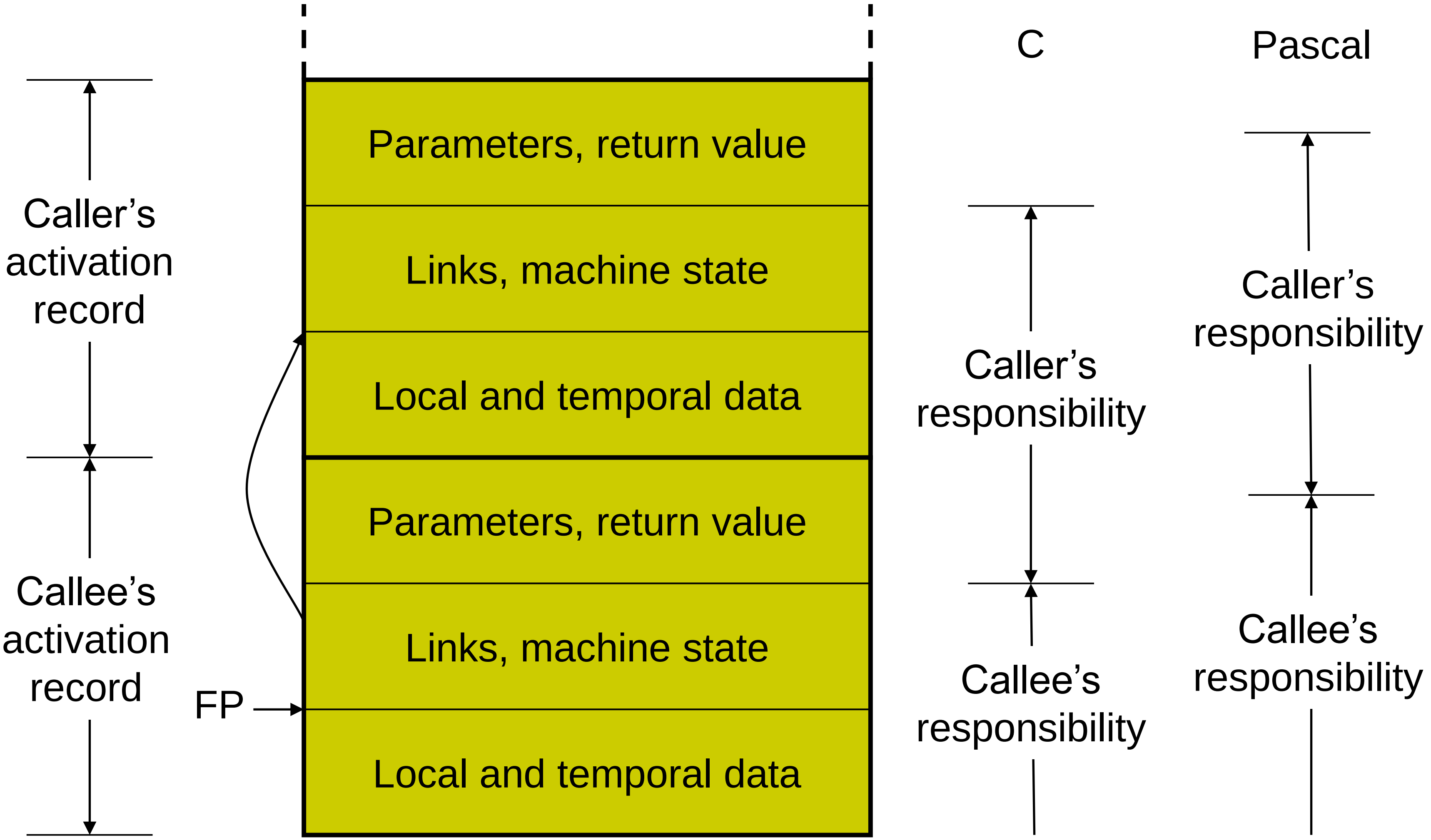
- Real meaning
  - mangle
    - mandlovat
    - rozsekat, roztrhat, rozbít, rozdrtit, těžce poškodit, potlouci, pohmožditi
    - *přen*. pokazit, znetvořit, k nepoznání změnit, překroutit, zkomolit
- Examples:

```
long f1(int i, const char *m, struct s *p)
```

`_f1`	MSVC IA-32 C __cdecl
`@f1@12`	MSVC IA-32 C __fastcall
`_f1@12`	MSVC IA-32 C __stdcall
`?f1@@YAJHPBDPAUs@@@Z`	MSVC IA-32 C++
`_f1`	GCC IA-32 C
`__Z2f1iPKcP1s`	GCC IA-32 C++
`f1`	MSVC IA-64 C
`?f1@@YAJHPEBDPEAUs@@@Z`	MSVC IA-64 C++



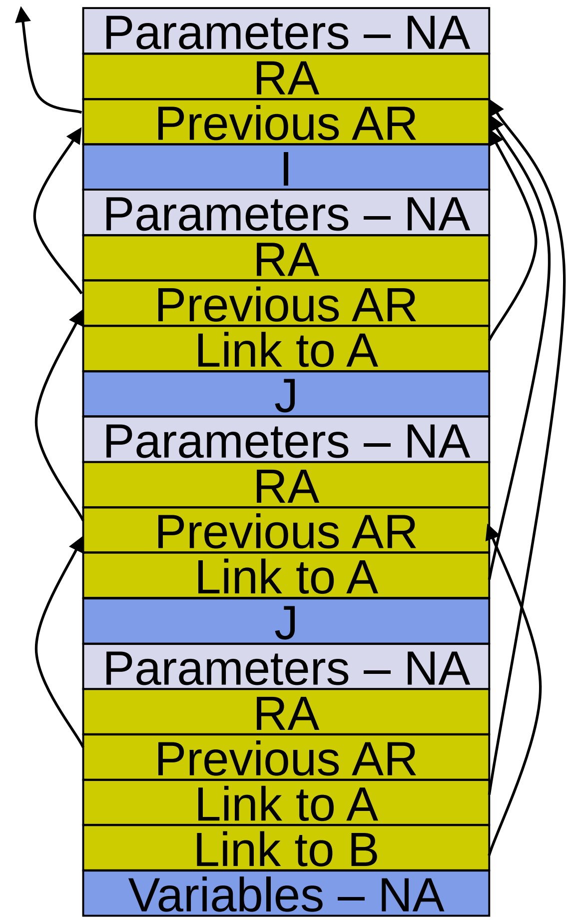
Call/return sequence



Access to semantically superior variables

- Pascal
  - Nested functions

```
procedure A;
var I:integer;
   procedure B;
   var J:integer;
       procedure C;
       begin …I+J… end;
   begin
       if I>0 B else C;
   end;
begin B end;
```



Parameter passing

- Call by value
  - Actual parameter is evaluated and the value is passed
  - Input parameters, the parameter is like a local variable
  - C, non-VAR parameters in Pascal
- Call by reference
  - The caller passes a pointer to the storage
  - Input/output parameters
  - & in C++, VAR parameters in Pascal
- Call by name
  - Like a macro – actual expression substituted at the point of use

Dynamic memory

- Allocation algorithms
  - Continuous blocks of variable length
- Garbage collector
  - Implicit deallocation
  - Reference counting
  - Markers
    - Stop the program at some point of execution
    - All pointers must be known including types
    - All blocks marked as unused
    - Go recursively through pointers and mark used blocks
    - Unused blocks are removed
    - Possible memory consolidation

Compiler principles

Interpreted languages

Jakub Yaghob

What are we talking about?

- The source language is not translated to a CPU code. Instead, it is translated to an abstract machine code
- Native compiled interpreter simulates the abstract machine

Why?

- No space for a compiler
  - 8-bit computers and BASIC
- Portability
  - The same abstract machine can run od different Oss and different CPU architectures
  - AS/400, Java
- Security
  - Better control of executed instructions

Problems

- Speed
  - It can be partially solved by JIT (Just-In-Time compilation)
    - Whenever the interpreter should execute abstract machine code, it is immediately compiled to native CPU instructions and stored in a cache
  - AoT (Ahead-of-Time)
    - Compile during installation
- Portability
  - Changing abstract machine behavior causes problems with portability
    - Java
- How to design the abstract machine/code
  - It should cover all source languages
    - .NET

Dynamic memory

- When supported, always with garbage collector
  - Pointers under control
  - Easy programming
  - Faster dynamic memory
    - Program usually does not need too much memory, therefore GC is not invoked at all and all memory is just quickly allocated
    - The abstract machine simulator usually takes more memory then would have