Compilers

Pierre Geurts

2016-2017

E-mail : p.geurts@ulg.ac.be
URL : http://www.montefiore.ulg.ac.be/~geurts/compil.html
Bureau : I 141 (Montefiore)
Téléphone : 04.366.48.15

Contact information

- Teacher: Pierre Geurts
  - p.geurts@ulg.ac.be, I141 Montefiore, 04/3664815
- Teaching assistant: Cyril Soldani
  - soldani@run.montefiore.ulg.ac.be, 1/10 B37, 04/3662699
- Website:
  - Course:
    http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2016/compil2016_2017.html
  - Project:
    http://www.montefiore.ulg.ac.be/~info0085

Course organization

- "Theoretical" course
  - Wednesday, 14h-16h, R75, Institut Montefiore
  - About 6-7 lectures
  - Slides online on the course web page
  - Give you the basis to achieve the project (and a little more)
- Project
  - One (big) project
  - Implementation of a compiler (from scratch) for a particular language.
  - A few repetition lectures on Wednesday, 16h-18h (checkpoints for your project).
  - (more on this later)
- Evaluation
  - Mostly on the basis of the project (but not only)
  - Written report, oral exam

References

- Books:
  - **Compilers: Principles, Techniques, and Tools (2nd edition), Aho, Lam, Sethi, Ullman, Prentice Hall, 2006**
    `http://dragonbook.stanford.edu/`
  - Modern compiler implementation in Java/C/ML, Andrew W. Appel, Cambridge University Press, 1998
    `http://www.cs.princeton.edu/~appel/modern/`
  - Engineering a compiler (2nd edition), Cooper and Torczon, Morgan Kaufmann, 2012.
- On the Web:
  - **Basics of compiler design, Torben Aegidius Mogensen, Self-published, 2010**
    `http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/index.html`
  - Compilation - Théorie des langages, Sophie Gire, Université de Brest
    `http://www.lisyc.univ-brest.fr/pages_perso/leparc/Etud/Master/Compil/Doc/CoursCompilation.pdf`
  - Standford compilers course
    `http://www.stanford.edu/class/cs143/`

Course outline

Part 1:	Introduction
Part 2:	Lexical analysis
Part 3:	Syntax analysis
Part 4:	Semantic analysis
Part 5:	Intermediate code generation
Part 6:	Code generation
Part 7:	Conclusion

Part 1

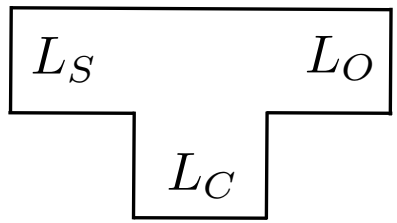
Introduction

Outline

1. What is a compiler
2. Compiler structure
3. Course project

Compilers

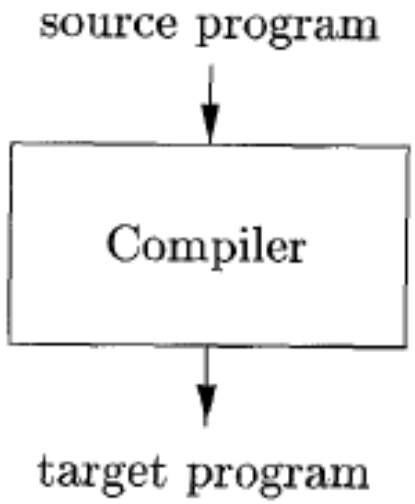
- A compiler is a program (written in a language L_c) that:
  - reads another program written in a given source language L_s
  - and translates (compiles) it into an equivalent program written in a second (target) language L_O.

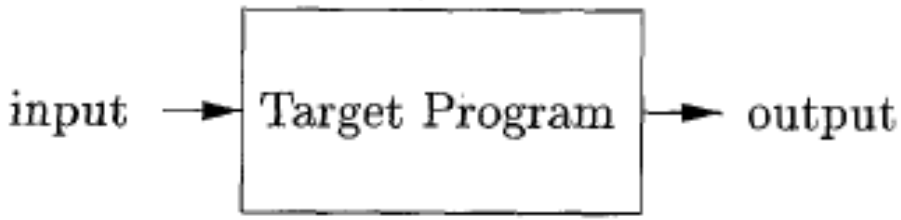


- The compiler also returns all errors contained in the source program

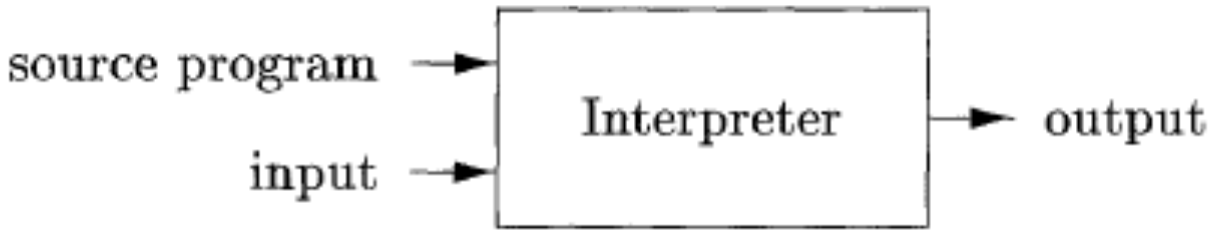
- Examples of combination:
  - L_C=C, L_S =C, L_O=Assembly (gcc)
  - L_C=C, L_S =java, L_O=C
  - L_C=java, L_S =LaTeX, L_O=HTML
  - ...
- Bootstrapping: $L_C = L_S$

Compiler


source program
Compiler
target program

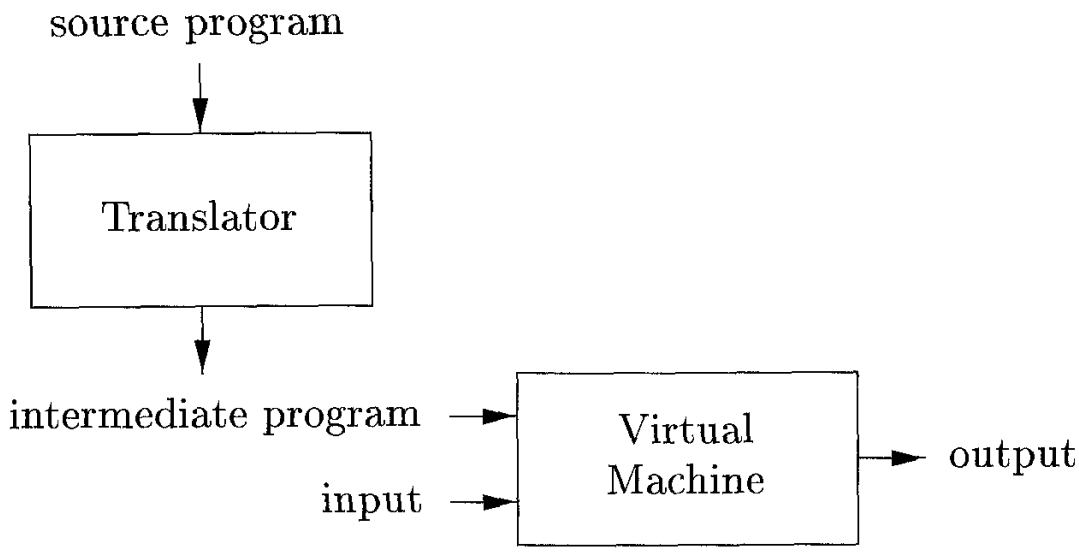

input
Target Program
output

Interpreter



- An interpreter is a program that:
  - executes directly the operations specified by the source program on input data provided by the user
- Usually slower at mapping inputs to outputs than compiled code (but gives better error diagnostics)

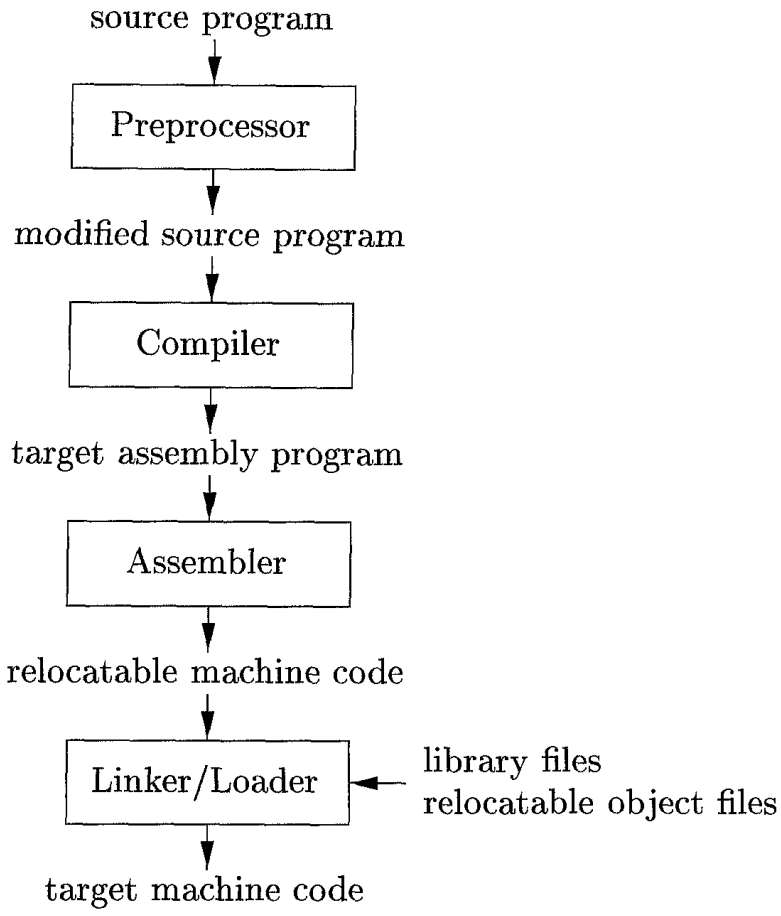
Hybrid solution



- Hybrid solutions are possible
- Example: Java combines compilation and interpretation
  - Java source program is compiled into an intermediate form called *bytecodes*
  - Bytecodes are then interpreted by a java virtual machine (or compiled into machine language by *just-in-time* compilers).
- Main advantage is portability

A broader picture

- Preprocessor: include files, macros... (small compiler).
- Assembler: generate machine code from assembly program (small trivial compiler).
- Linker: relocates relative addresses and resolves external references.
- Loader: loads the executable file in memory for execution.



Why study compilers?

- There is small chance that you will ever write a full compiler in your professional carrier.
- Then why study compilers?
  - To improve your culture in computer science (not a very good reason)
  - To get a better intuition about high-level languages and therefore become a better coder
  - Compilation is not restricted to the translation of computer programs into assembly code
    - Translation between two high-level languages (Java to C++, Lisp to C, Python to C, etc.)
    - Translation between two arbitrary languages, not necessarily programming ones (word to html, pdf to ps, etc.), aka source-to-source compilers or transcompilers

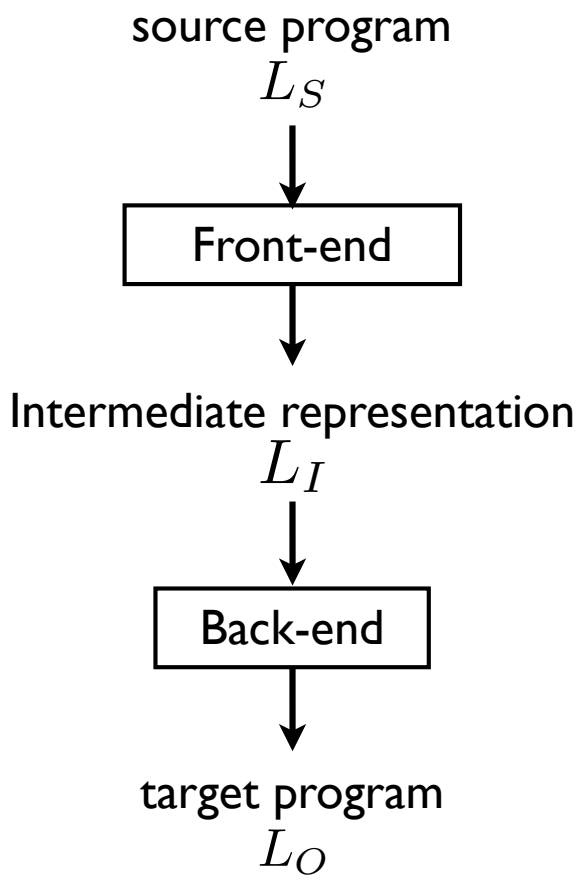
Why study compilers?

- The techniques behind compilers are useful for other purposes as well
  - Data structures, graph algorithms, parsing techniques, language theory...
- There is a good chance that a computer scientist will need to write a compiler or an interpreter for a domain-specific language
  - Example: database query languages, text-formatting language, scene description language for ray-tracers, search engine, sed/awk, substitution in parameterized code...
- Very nice application of concepts learned in other courses
  - Data structures and algorithms, introduction to the theory of computation, computation structures...

General structure of a compiler

- Except in very rare cases, translation can not be done word by word
- Compilers are (now) very structured programs
- Typical structure of a compiler in two stages:
  - Front-end/analysis:
    - Breaks the source program into constituent pieces
    - Detect syntaxic and semantic errors
    - Produce an intermediate representation of the language
    - Store in a symbol table information about procedures and variables of the source program
  - Back-end/synthesis:
    - Construct the target program from the intermediate representation and the symbol table
  - Typically, the front end is independent of the target language, while the back end is independent of the source language
  - One can have a middle part that optimizes the intermediate representation (and is thus independent of both the source and target languages)

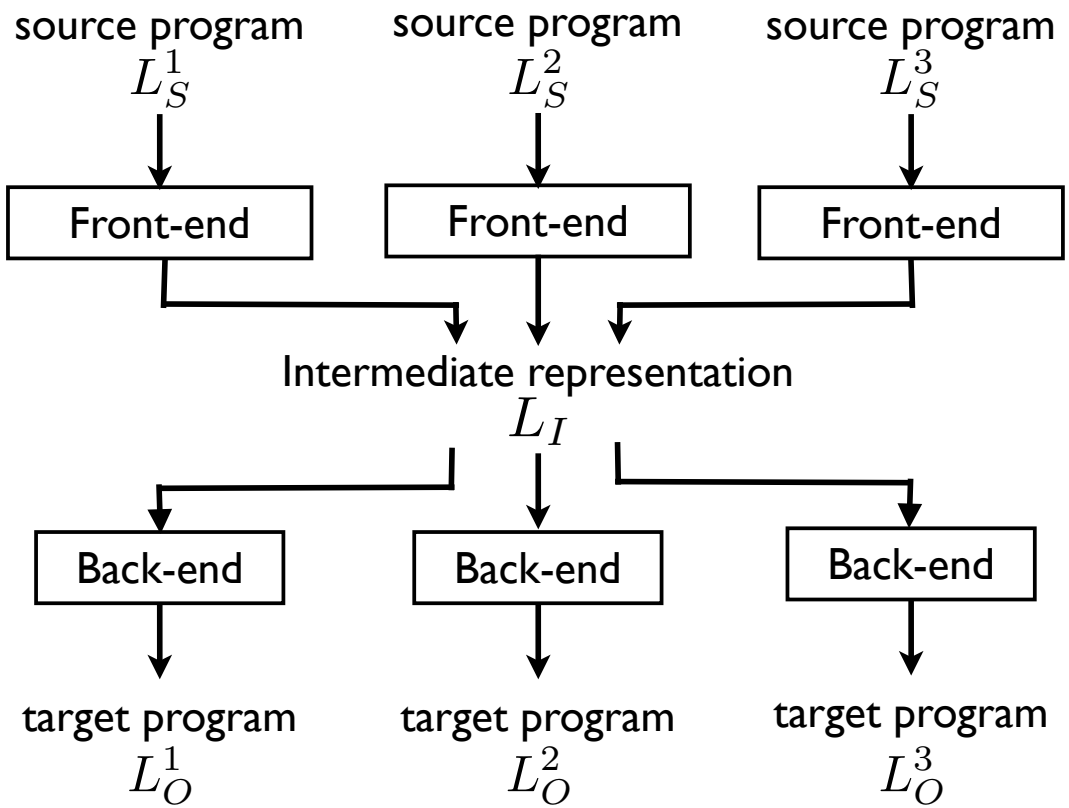
General structure of a compiler



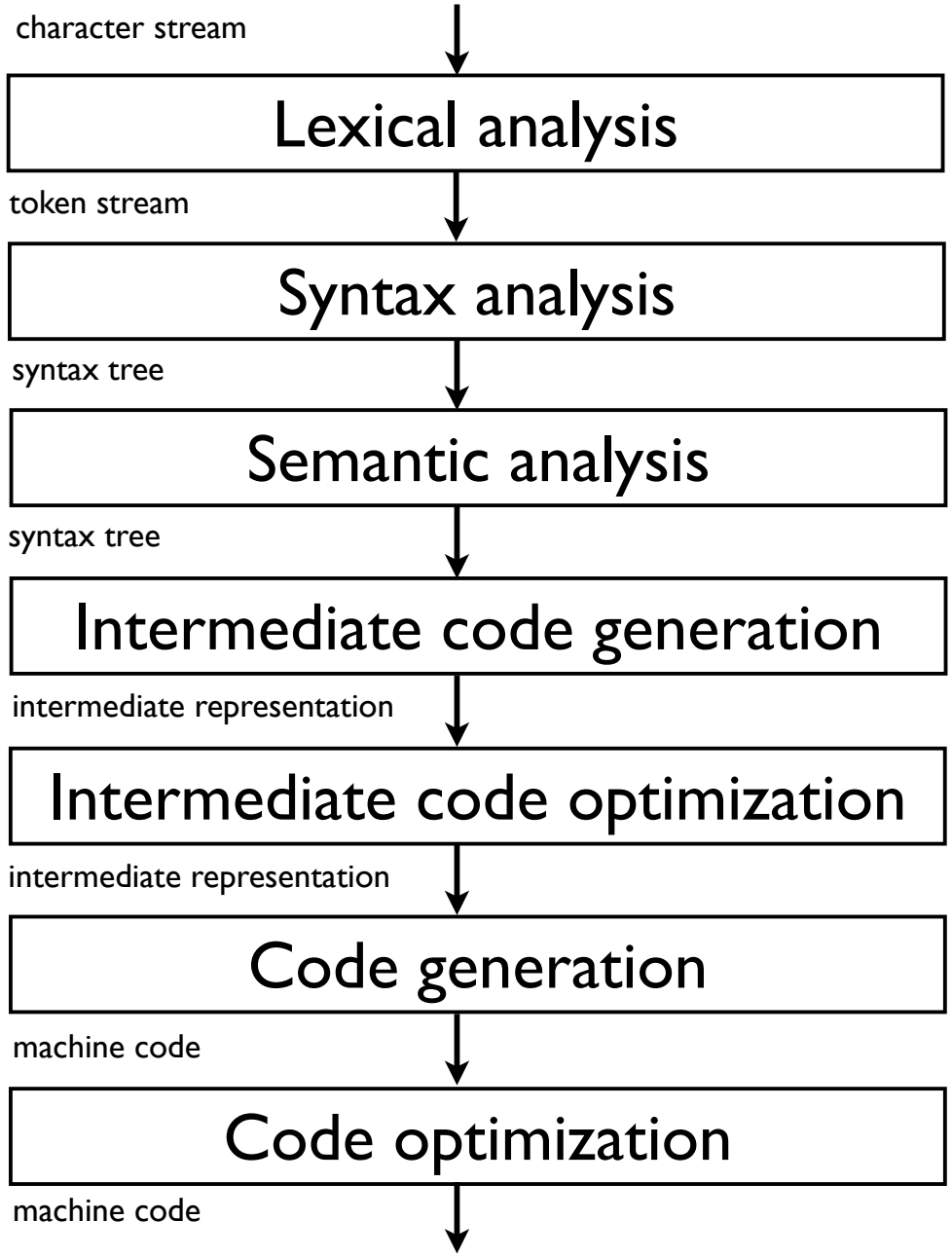
Intermediate representation

The intermediate representation:

- Ensures portability (it's easy to change the source or the target language by adapting the front-end or back-end).
- Should be at the same time easy to produce from the source language and easy to translate into the target language



Detailed structure of a compiler



Lexical analysis or scanning

Input: Character stream $\Rightarrow$ **Output:** token streams

- The lexical analyzer groups the characters into meaningful sequences called lexemes.
  - Example: "`position = initial + rate * 60;`" is broken into the lexemes `position`, `=`, `initial`, `+`, `rate`, `*`, 60, and `;`.
  - (Non-significant blanks and comments are removed during scanning)
- For each lexeme, the lexical analyzer produces as output a token of the form: $\langle \textit{token-name}, \textit{attribute-value} \rangle$
  - The produced tokens for "`position = initial + rate * 60`" are as follows

$$\langle \textbf{id}, 1 \rangle, \langle \textbf{op}, = \rangle, \langle \textbf{id}, 2 \rangle, \langle \textbf{op}, + \rangle, \langle \textbf{id}, 3 \rangle, \langle \textbf{op}, * \rangle, \langle \textbf{num}, 60 \rangle$$

with the symbol table:

1	position	...
2	initial	...
3	rate	...

(In modern compilers, the table is not built anymore during lexical analysis)

Lexical analysis or scanning

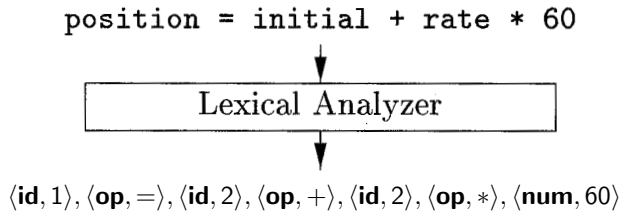
In practice:

- Each token is defined by a regular expression
  - Example:
    $Letter = A - Z|a - z$
    $Digit = 0 - 9$
    $Identifier = Letter(Letter|Digit)^*$

- Lexical analysis is implemented by
  - building a non deterministic finite automaton from all token regular expressions
  - eliminating non determinism
  - Simplifying it

- There exist automatic tools to do that
  - Examples: lex, flex...

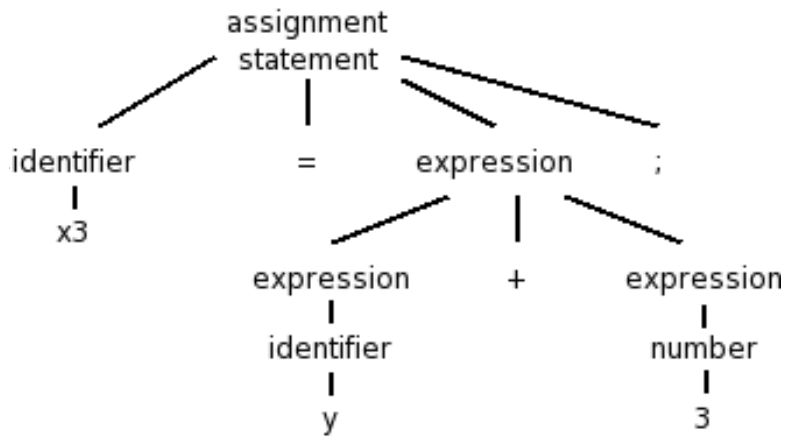
Lexical analysis or scanning



Syntax analysis or parsing

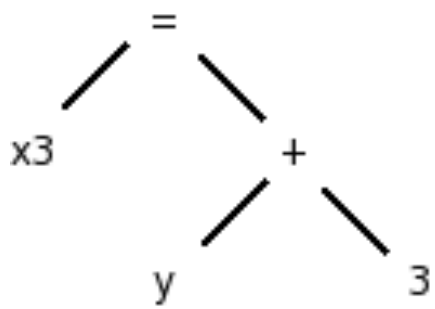
Input: token stream $\Rightarrow$ **Output:** syntax tree

- Parsing groups tokens into grammatical phrases
- The result is represented in a parse tree, ie. a tree-like representation of the grammatical structure of the token stream.
- Example:
  - Grammar for assignement statement:
    asst-stmt $\rightarrow$ id $=$ exp ;
    exp $\rightarrow$ number | id | expr $+$ expr
  - Example parse tree:



Syntax analysis or parsing

- The parse tree is often simplified into a (abstract) syntax tree:



- This tree is used as a base structure for all subsequent phases

- On parsing algorithms:
  - Languages are defined by context-free grammars
  - Parse and syntax trees are constructed by building automatically a (kind of) pushdown automaton from the grammar
  - Typically, these algorithms only work for a (large) subclass of context-free grammars

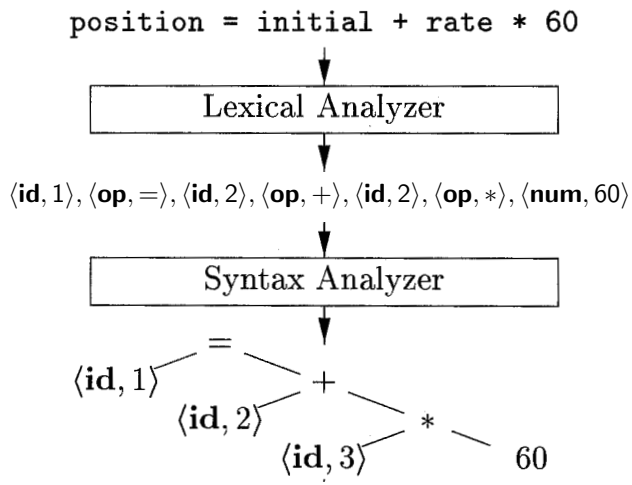
Lexical versus syntax analysis

- The division between scanning and parsing is somewhat arbitrary.
- Regular expressions could be represented by context-free grammars
- Mathematical expression grammar:

	EXPRESSION	$\rightarrow$	EXPRESSION OP2 EXPRESSION
Syntax	EXPRESSION	$\rightarrow$	NUMBER
	EXPRESSION	$\rightarrow$	(EXPRESSION)
	OP2	$\rightarrow$	$+ \mid - \mid * \mid /$
Lexical	NUMBER	$\rightarrow$	DIGIT | DIGIT NUMBER
	DIGIT	$\rightarrow$	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- The main goal of lexical analysis is to simplify the syntax analysis (and the syntax tree).

Syntax analysis or parsing



Semantic analysis

Input: syntax tree $\Rightarrow$ **Output:** (augmented) syntax tree

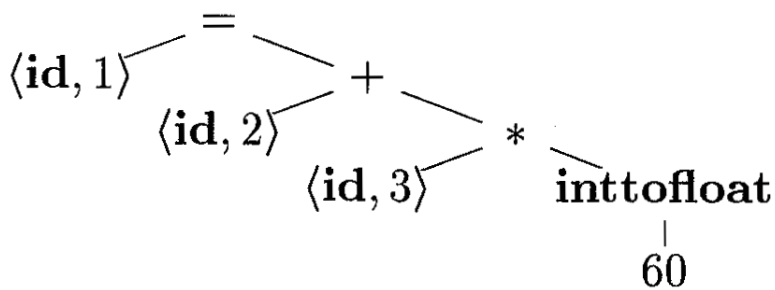
- Context-free grammar can not represent all language constraints, e.g. non local/context-dependent relations.

- Semantic/contextual analysis checks the source program for semantic consistency with the language definition.
  - A variable can not be used without having been defined
  - The same variable can not be defined twice
  - The number of arguments of a function should match its definition
  - One can not multiply a number and a string
  - ...

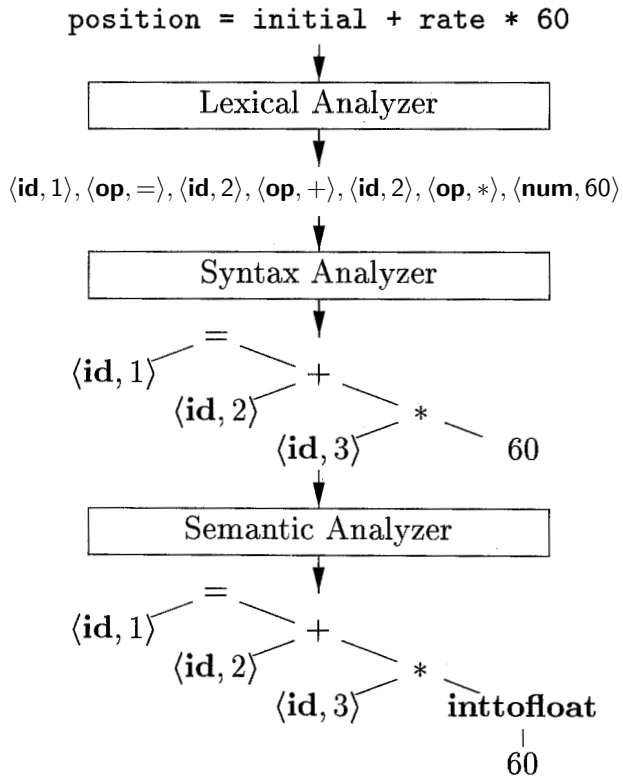
  (none of these constraints can be represented in a context-free grammar)

Semantic analysis

- Semantic analysis also carries out type checking:
  - Each operator should have matching operands
  - In some cases, type conversions (coercions) might be possible (e.g., for numbers)
- Example: `position = initial + rate * 60`
  If the variables `position`, `initial`, and `rate` are defined as floating-point variables and 60 was read as an integer, it may be converted into a floating-point number.



Semantic analysis



Intermediate code generation

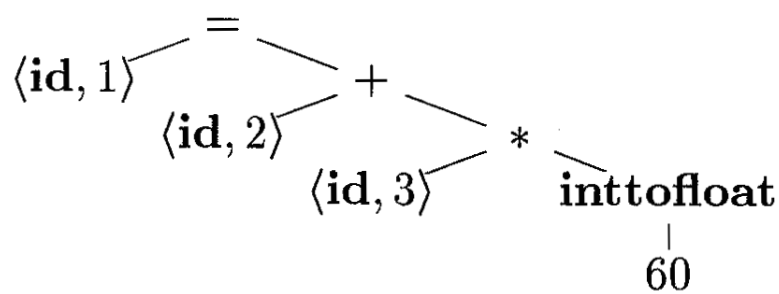
Input: syntax tree $\Rightarrow$ **Output:** Intermediate representation

- A compiler typically uses one or more intermediate representations
  - Syntax trees are a form of intermediate representation used for syntax and semantic analysis
- After syntax and semantic analysis, many compilers generate a low-level or machine-like intermediate representation
- Two important properties of this intermediate representation:
  - Easy to produce
  - Easy to translate into the target machine code

Intermediate code generation

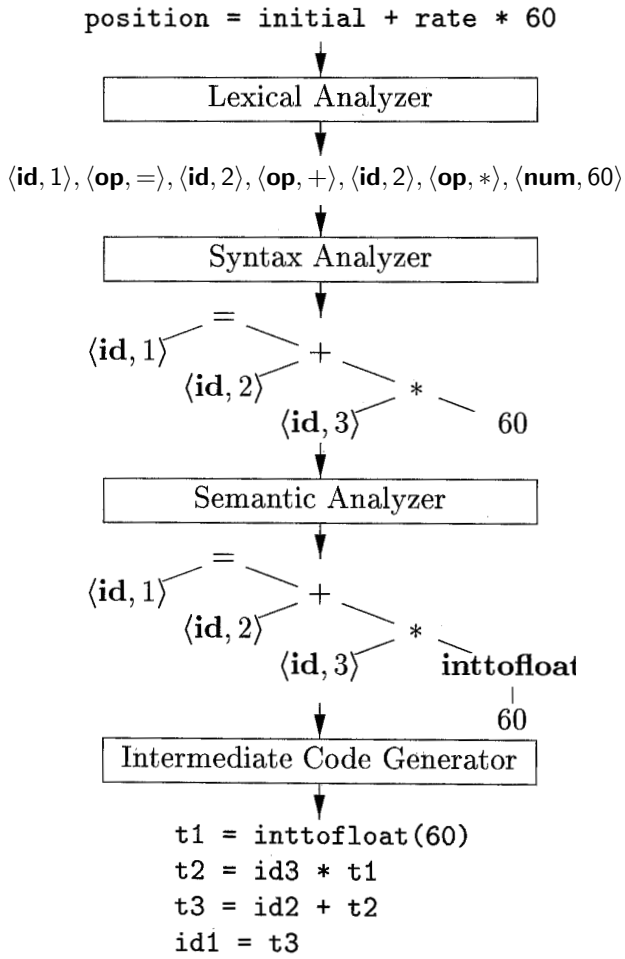
- Example: Three-address code with instructions of the form `x = y op z`.
  - Assembly-like instructions with three operands (at most) per instruction
  - Assumes an unlimited number of registers

- Translation of the syntax tree




```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Intermediate code generation



Intermediate code optimization

Input: Intermediate representation $\Rightarrow$ **Output:** (better) intermediate representation

- Goal: improve the intermediate code (to get better target code at the end)
- Machine-independent optimization (versus machine-dependent optimization of the final code)
- Different criteria: efficiency, code simplicity, power consumption...
- Example:

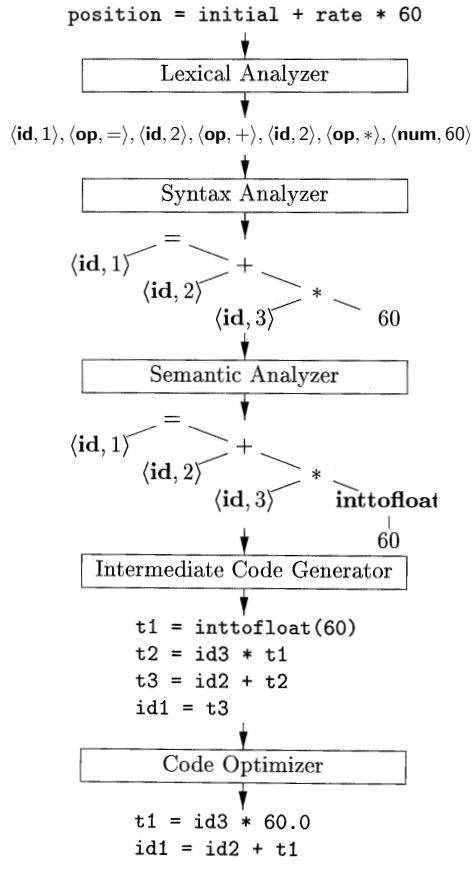
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

$\Rightarrow$

```
t1 = id3 * 60.0
id1 = id2 + t1
```

- Optimization is complex and very time consuming
- Very important step in modern compilers

Intermediate code optimization



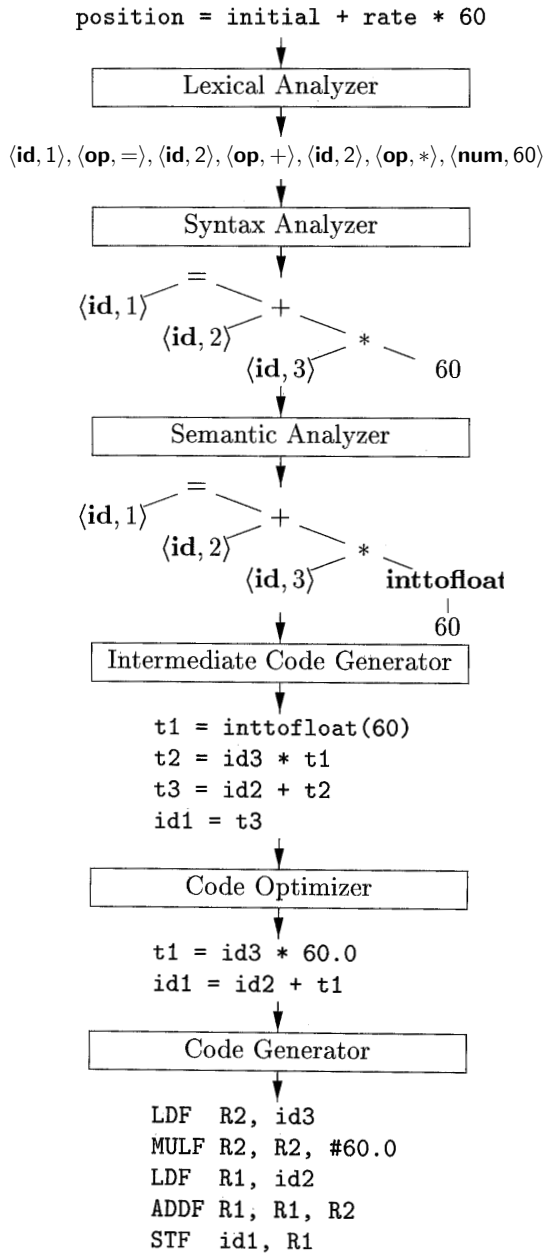
Code generation

Input: Intermediate representation $\Rightarrow$ **Output:** target machine code

- From the intermediate code to real assembly code for the target machine
- Needs to take into account specifities of the target machine, eg., number of registers, operators in instruction, memory management.
- One crucial aspect is register allocation
- For our example:

```
t1 = id3 * 60.0
id1 = id2 + t1
⇒
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1,R1
```

Final code generation



Symbol table

1	position	...
2	initial	...
3	rate	...

- Records all variable names used in the source program
- Collects information about each symbol:
  - Type information
  - Storage location (of the variable in the compiled program)
  - Scope
  - For function symbol: number and types of arguments and the type returned
- Built during lexical analysis (old way) or in a separate phase (modern way).
- Needs to allow quick retrieval and storage of a symbol and its attached information in the table
- Implementation by a dictionary structure (binary search tree, hash-table,...).

Error handling

- Each phase may produce errors.
- A good compiler should report them and provide as much information as possible to the user.
  - Not only “syntax error”.
- Ideally, the compiler should not stop after the first error but should continue and detect several errors at once (to ease debugging).

Phases and Passes

- The description of the different phases makes them look sequential
- In practice, one can combine several phases into one pass (i.e., one complete reading of an input file or traversal of the intermediate structures).
- For example:
  - One pass through the initial code for lexical analysis, syntax analysis, semantic analysis, and intermediate code generation (front-end).
  - One or several passes through the intermediate representation for code optimization (optional)
  - One pass through the intermediate representation for the machine code generation (back-end)

Compiler-construction tools

- First compilers were written from scratch, and considered as very difficult programs to write.
  - The first fortran compiler (IBM, 1957) required 18 man-years of work
- There exist now several theoretical tools and softwares to automate several phases of the compiler.
  - Lexical analysis: regular expressions and finite state automata (Software: (f)lex)
  - Syntax analysis: grammars and pushdown automata (Softwares: bison/yacc, ANTLR)
  - Semantic analysis and intermediate code generation: syntax directed translation
  - Code optimization: data flow analysis

This course

- Although the back-end is more and more important in modern compilers, we will insist more on the front-end and general principles
- Outline:
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate code generation (syntax directed translation)
  - Some notions about code generation and optimization

Compiler project

(Subject to changes)

- Implement a "complete" compiler
- By group of 1, **2**, or 3 students
- Source language: VSOP, The Very Simple Object-oriented Programming language (by Cyril Soldani)
  - An academic object-oriented programming language inspired by COOL (the Classroom Object-Oriented Language)
  - `http://www.montefiore.ulg.ac.be/~info0085/`
- The destination language will be LLVM, a popular modern intermediate language
  - `http://llvm.org/`
- Implementation language L_c can be chosen among c, c++, java, python, ocaml, scheme (for other languages, ask us).

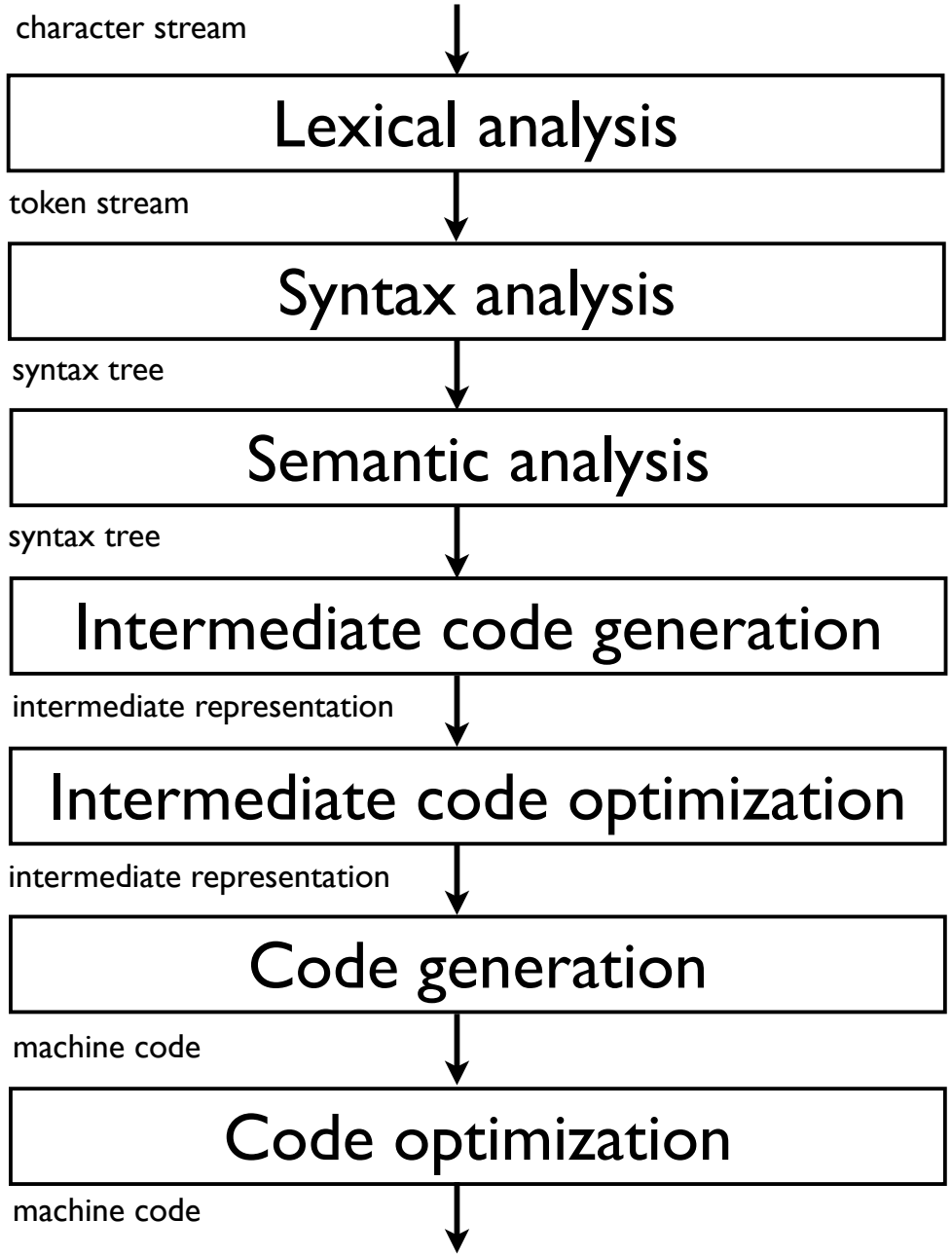
Part 2

Lexical analysis

Outline

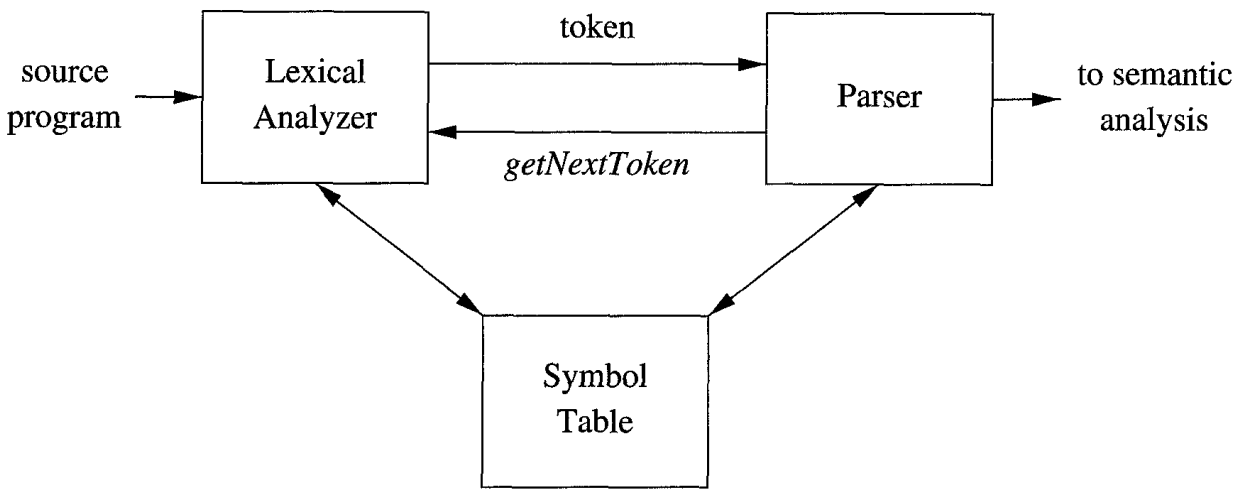
1. Principle
2. Regular expressions
3. Analysis with non-deterministic finite automata
4. Analysis with deterministic finite automata
5. Implementing a lexical analyzer

Structure of a compiler



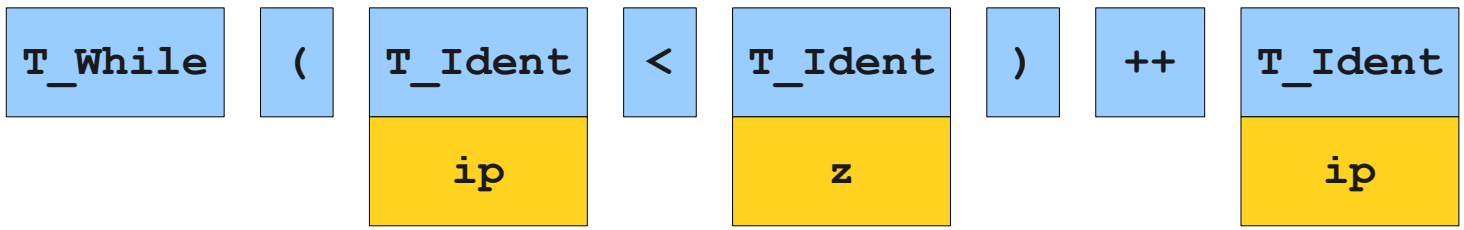
Lexical analysis or scanning

- Goals of the lexical analysis
  - Divide the character stream into meaningful sequences called lexemes.
  - Label each lexeme with a token that is passed to the parser (syntax analysis)
  - Remove non-significant blanks and comments
  - Optional: update the symbol tables with all identifiers (and numbers)
- Provide the interface between the source program and the parser



(Dragonbook)

Example

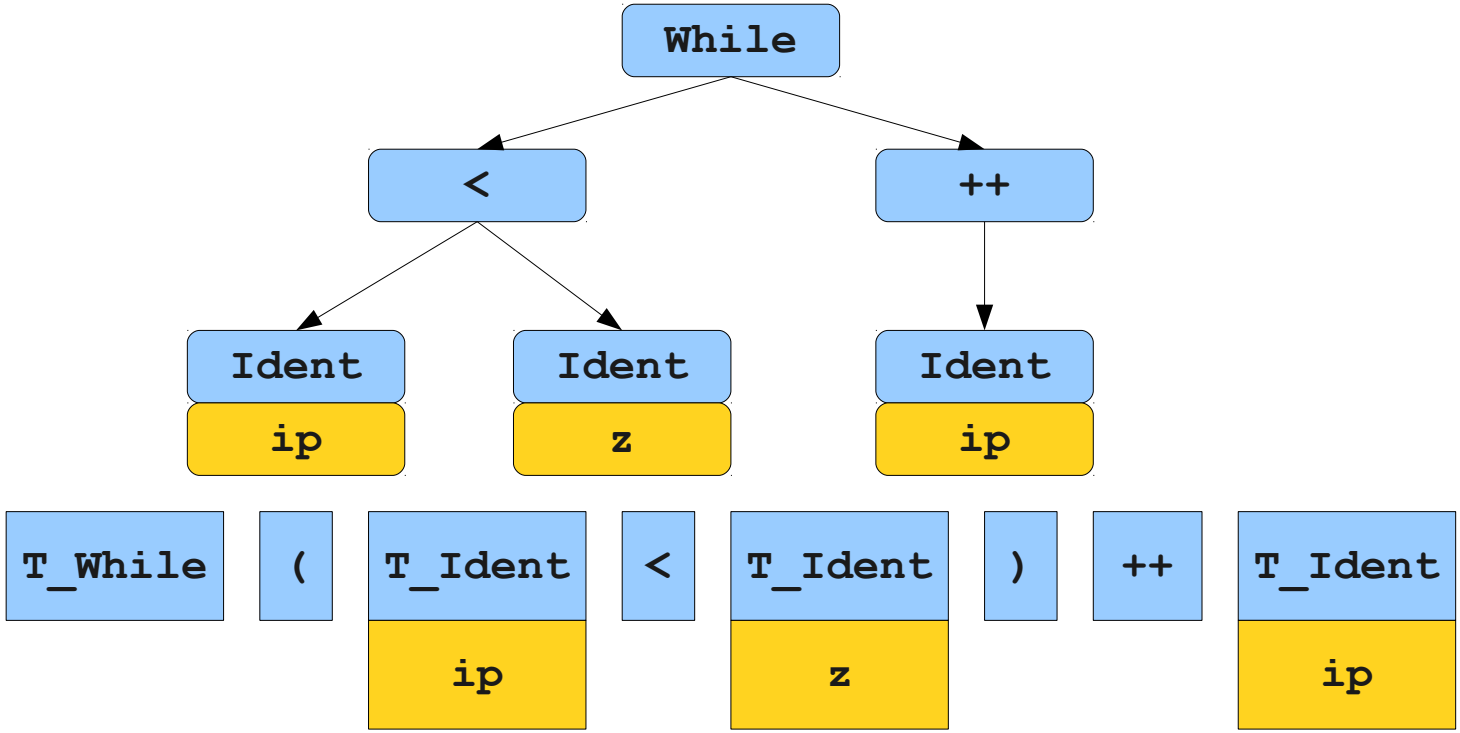


w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Example



w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Lexical versus syntax analysis

Why separate lexical analysis from parsing?

- Simplicity of design: simplify both the lexical analysis and the syntax analysis.
- Efficiency: specialized techniques can be applied to improve lexical analysis.
- Portability: only the scanner needs to communicate with the outside

Tokens, patterns, and lexemes

- A token is a $\langle name, attribute \rangle$ pair. Attribute might be multi-valued.
  - Example: $\langle Ident, ip \rangle$, $\langle Operator, < \rangle$, $\langle ')', NIL \rangle$
- A pattern describes the character strings for the lexemes of the token.
  - Example: a string of letters and digits starting with a letter, $\{<, >, \leq, \geq, ==\}$, ")".
- A lexeme for a token is a sequence of characters that matches the pattern for the token
  - Example: **ip**, "<", ")" in the following program

```
while (ip < z)
    ++ip
```

Defining a lexical analysis

1. Define the set of tokens
2. Define a pattern for each token (ie., the set of lexemes associated with each token)
3. Define an algorithm for cutting the source program into lexemes and outputting the tokens

Choosing the tokens

- Very much dependent on the source language
- Typical token classes for programming languages:
  - One token for each keyword
  - One token for each "punctuation" symbol (left and right parentheses, comma, semicolon...)
  - One token for identifiers
  - Several tokens for the operators
  - One or more tokens for the constants (numbers or literal strings)
- Attributes
  - Allows to encode the lexeme corresponding to the token when necessary. Example: pointer to the symbol table for identifiers, constant value for constants.
  - Not always necessary. Example: keyword, punctuation...

Describing the patterns

- A pattern defines the set of lexemes corresponding to a token.
- A lexeme being a string, a pattern is actually a language.
- Patterns are typically defined through regular expressions (that define regular languages).
  - Sufficient for most tokens
  - Lead to efficient scanner

Reminder: languages

- An alphabet Σ is a set of characters

  Example: $\Sigma = \{a, b\}$

- A string over Σ is a finite sequence of elements from Σ

  Example: $aabba$

- A language is a set of strings

  Example: $L = \{a, b, abab, babbba\}$

- Regular languages: a subset of all languages that can be defined by regular expressions

Reminder: regular expressions

- Any character $a \in \Sigma$ is a regular expression $L = \{a\}$
- ϵ is a regular expression $L = \{\epsilon\}$
- If R_1 and R_2 are regular expressions, then
  - $R_1 R_2$ is a regular expression
    $L(R_1 R_2)$ *is the concatenation of* $L(R1)$ *and* $L(R2)$
  - $R_1|R_2$ $(= R_1 \bigcup R_2)$ is a regular expression
    $L(R_1|R_2) = L(R_1) \bigcup L(R_2)$
  - R_1^* is a regular expression
    $L(R_1^*)$ *is the Kleene closure of* $L(R_1)$
  - (R_1) is a regular expression
    $L((R_1)) = L(R_1)$

- Example: a regular expression for even numbers:

$$(+|-|\epsilon)(0|1|2|3|4|5|6|7|8|9)^*(0|2|4|6|8)$$

Notational conveniences

- Regular definitions:

$$\begin{aligned} \textit{letter} &\rightarrow \text{A}|\text{B}|...|\text{Z}|\text{a}|\text{b}|...|\text{z} \\ \textit{digit} &\rightarrow 0|1|...|9 \\ \textit{id} &\rightarrow \textit{letter}(\textit{letter}|\textit{digit})^* \end{aligned}$$

- One or more instances: $r^+ = rr^*$
- Zero or one instance: $r? = r|\epsilon$
- Character classes:

$$\begin{aligned} [\text{abc}] &= \text{a}|\text{b}|\text{c} \\ [\text{a-z}] &= \text{a}|\text{b}|...|\text{z} \\ [0\text{-}9] &= 0|1|...|9 \end{aligned}$$

Examples

- Keywords:

  if, while, for, ...

- Identifiers:

  [a-zA-Z_][a-zA-Z_0-9]*

- Integers:

  [+−]?[0-9]+

- Floats:

  [+−]?(([0-9]+ (.[0-9]*)?|.[0-9]+)([eE][+−]?[0-9]+)?)

- String constants:

  “([a-zA-Z0-9]|\[a-zA-Z])*”

Algorithms for lexical analysis

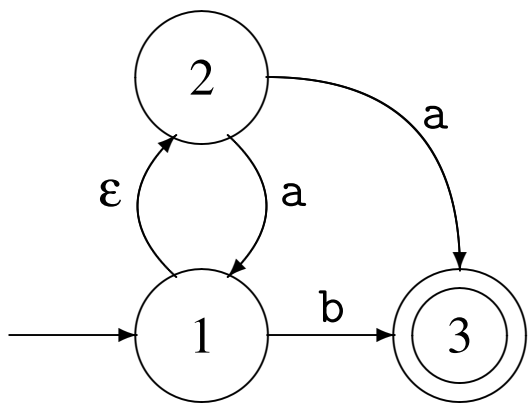
- How to perform lexical analysis from token definitions through regular expressions?
- Regular expressions are equivalent to finite automata, deterministic (DFA) or non-deterministic (NFA).
- Finite automata are easily turned into computer programs
- Two methods:
  1. Convert the regular expressions to an NFA and simulate the NFA
  2. Convert the regular expressions to an NFA, convert the NFA to a DFA, and simulate the DFA.

Reminder: non-deterministic automata (NFA)

A non-deterministic automaton is a five-tuple $M = (Q, \Sigma, \Delta, s_0, F)$ where:

- Q is a finite set of states,
- Σ is an alphabet,
- $\Delta \subset (Q \times (\Sigma \bigcup \{\epsilon\}) \times Q)$ is the transition relation,
- $s \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states

Example:



Transition table

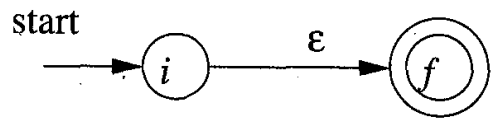
State	a	b	ϵ
1	$\emptyset$	{3}	{2}
2	{1,3}	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

(Mogensen)

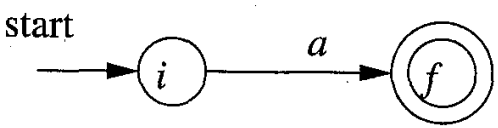
Reminder: from regular expression to NFA

A regular expression can be transformed into an equivalent NFA

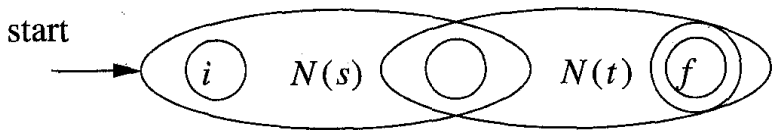
ϵ



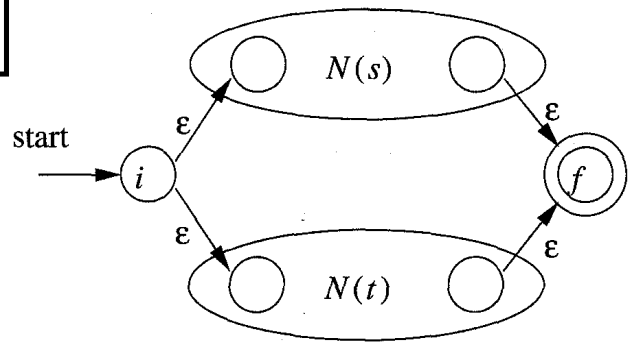
a



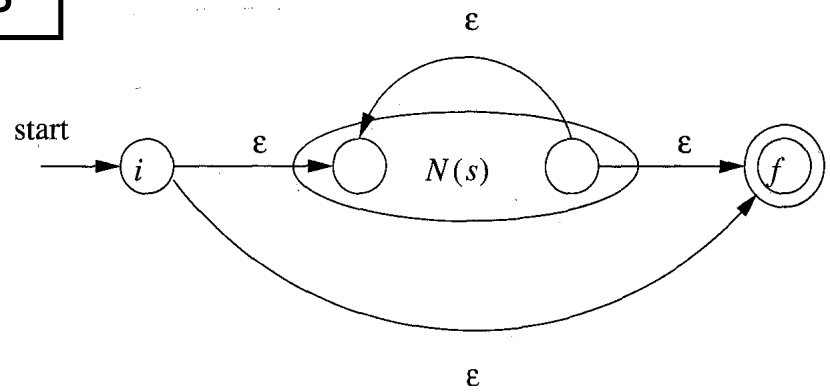
st



s|t



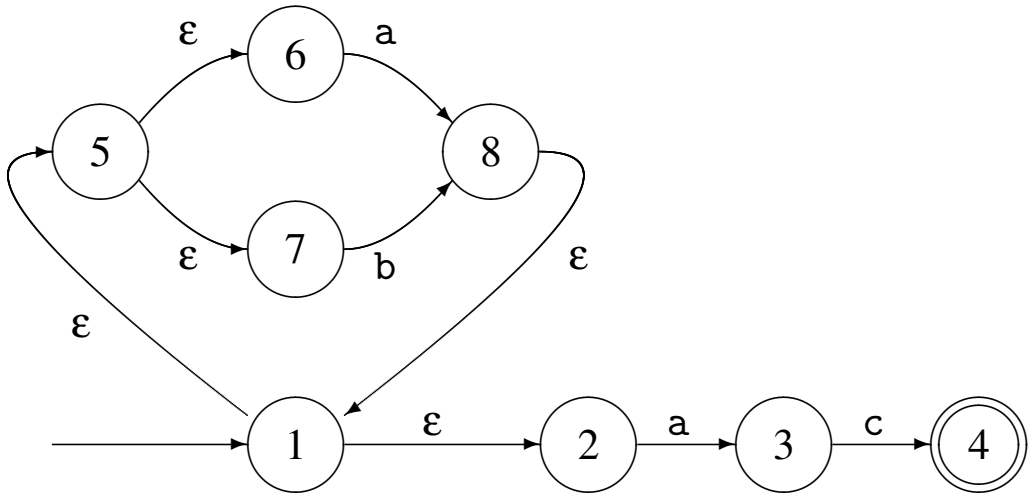
s*



(Dragonbook)

Reminder: from regular expression to NFA

Example: (a|b)*ac (Mogensen)



The NFA $N(r)$ for an expression r is such that:

- $N(r)$ has at most twice as many states as there are operators and operands in R.
- $N(r)$ has one initial state and one accepting state (with no outgoing transition from the accepting state and no incoming transition to the initial state).
- Each (non accepting) state in $N(r)$ has either one outgoing transition or two outgoing transitions, both on ϵ.

Simulating an NFA

Algorithm to check whether an input string is accepted by the NFA:

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

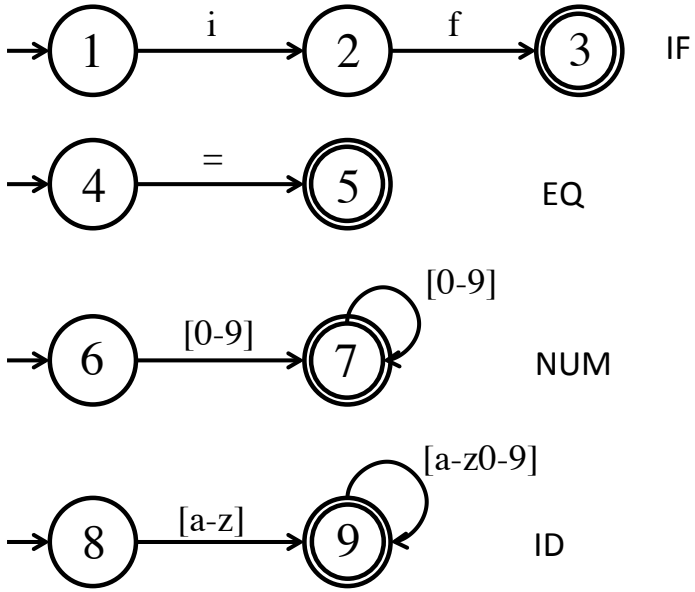
(Dragonbook)

- *nextChar*(): returns the next character on the input stream
- *move*(S, c): returns the set of states that can be reached from states in S when observing c.
- ϵ-*closure*(S): returns all states that can be reached with ϵ transitions from states in S.

Lexical analysis

- What we have so far:
  - Regular expressions for each token
  - NFAs for each token that can recognize the corresponding lexemes
  - A way to simulate an NFA
- How to combine these to cut apart the input text and recognize tokens?
- Two ways:
  - Simulate all NFAs in turn (or in parallel) from the current position and output the token of the first one to get to an accepting state
  - Merge all NFAs into a single one with labels of the tokens on the accepting states

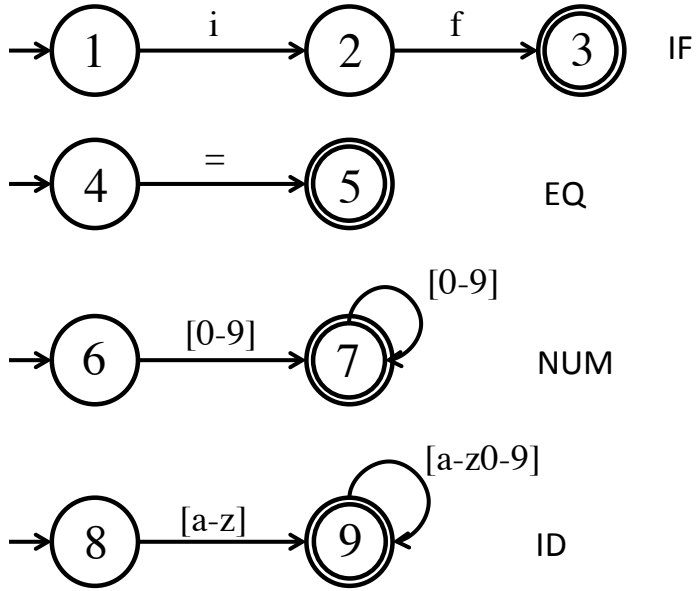
Illustration



- Four tokens: IF=if, ID=[a-z][a-z0-9]*, EQ='=', NUM=[0-9]$^+$
- Lexical analysis of $x = 6$ yields:

$$\langle ID, x \rangle, \langle EQ \rangle, \langle NUM, 6 \rangle$$

Illustration: ambiguities



- Lexical analysis of $ifu26 = 60$
- Many splits are possible:

$$\langle IF \rangle, \langle ID, u26 \rangle, \langle EQ \rangle, \langle NUM, 60 \rangle$$

$$\langle ID, ifu26 \rangle, \langle EQ \rangle, \langle NUM, 60 \rangle$$

$$\langle ID, ifu \rangle, \langle NUM, 26 \rangle, \langle EQ \rangle, \langle NUM, 6 \rangle, \langle NUM, 0 \rangle$$

....

Conflict resolutions

- Principle of the longest matching prefix: we choose the longest prefix of the input that matches any token
- Following this principle, $ifu26 = 60$ will be split into:

$$\langle ID, ifu26 \rangle, \langle EQ \rangle, \langle NUM, 60 \rangle$$

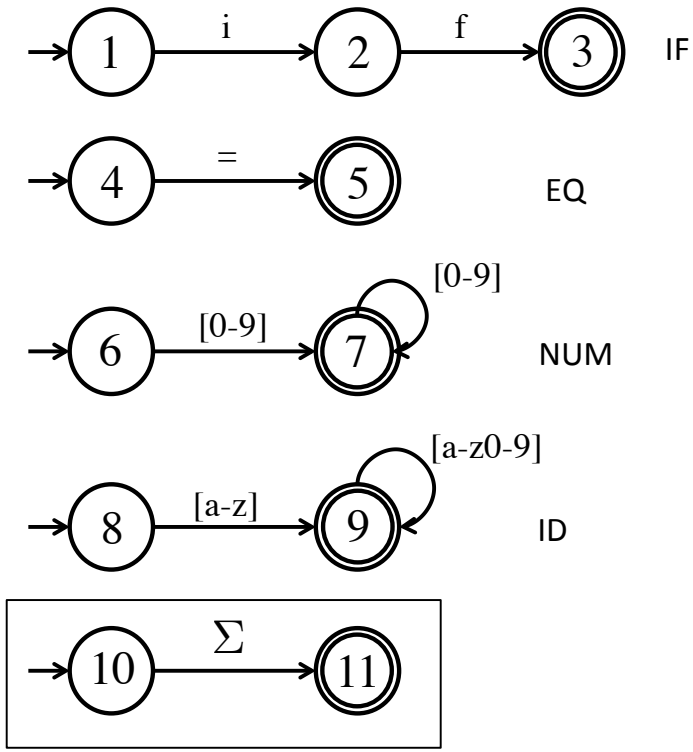
- How to implement?
  - Run all NFAs in parallel, keeping track of the last accepting state reached by any of the NFAs
  - When all automata get stuck, report the last match and restart the search at that point
- Requires to retain the characters read since the last match to re-insert them on the input
  - In our example, '=' would be read and then re-inserted in the buffer.

Other source of ambiguity

- A lexeme can be accepted by two NFAs
  - Example: keywords are often also identifiers (*if* in the example)
- Two solutions:
  - Report an error (such conflict is not allowed in the language)
  - Let the user decide on a priority order on the tokens (eg., keywords have priority over identifiers)

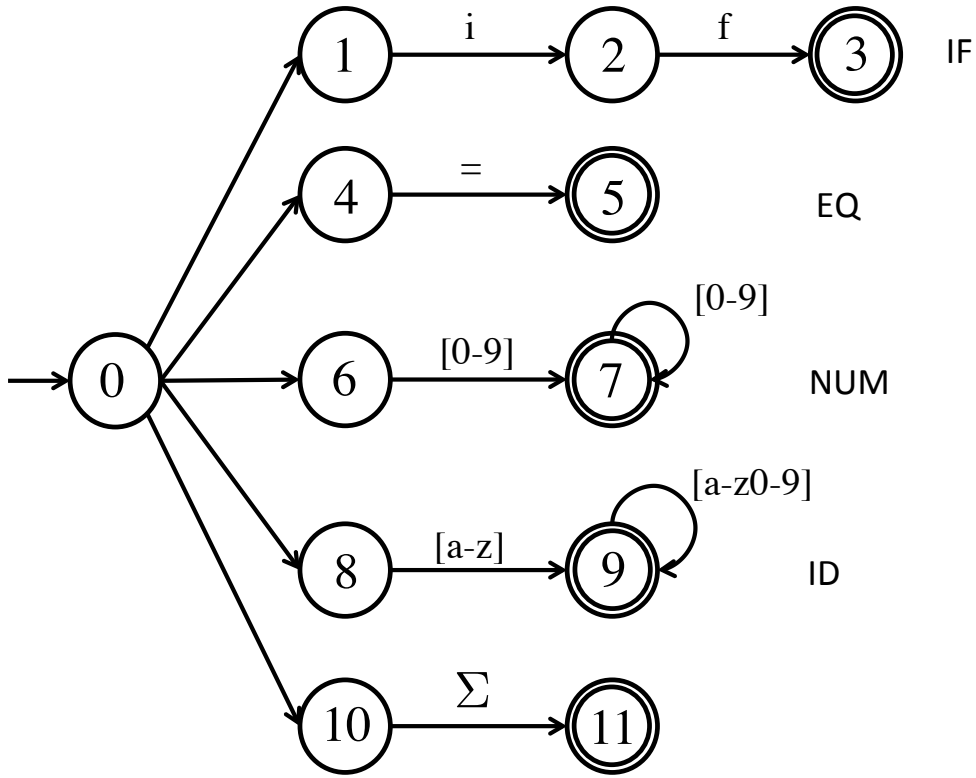
What if nothing matches

- What if we can not reach any accepting states given the current input?
- Add a "catch-all" rule that matches any character and reports an error



Merging all automata into a single NFA

- In practice, all NFAs are merged and simulated as a single NFA
- Accepting states are labeled with the token name



Lexical analysis with an NFA: summary

- Construct NFAs for all regular expressions
- Merge them into one automaton by adding a new start state
- Scan the input, keeping track of the last known match
- Break ties by choosing higher-precedence matches
- Have a catch-all rule to handle errors

Computational efficiency

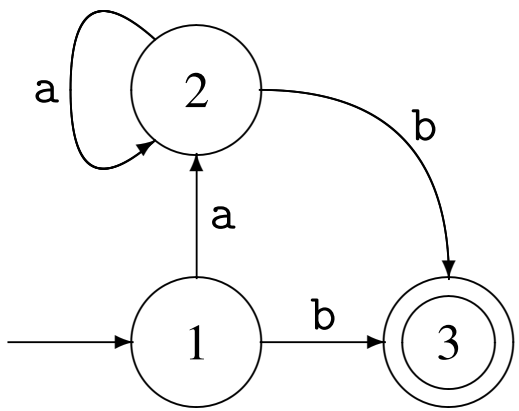
```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

(Dragonbook)

- In the worst case, an NFA with $|Q|$ states takes $O(|S||Q|^2)$ time to match a string of length $|S|$
- Complexity thus depends on the number of states
- It is possible to reduce complexity of matching to $O(|S|)$ by transforming the NFA into an equivalent deterministic finite automaton (DFA)

Reminder: deterministic finite automaton

- Like an NFA but the transition relation $\Delta \subset (Q \times (\Sigma \bigcup \{\epsilon\}) \times Q)$ is such that:
  - Transitions based on ϵ are not allowed
  - Each state has at most one outgoing transition defined for every letter
- Transition relation is replaced by a transition function $\delta : Q \times \Sigma \to Q$
- Example of a DFA



(Mogensen)

Reminder: from NFA to DFA

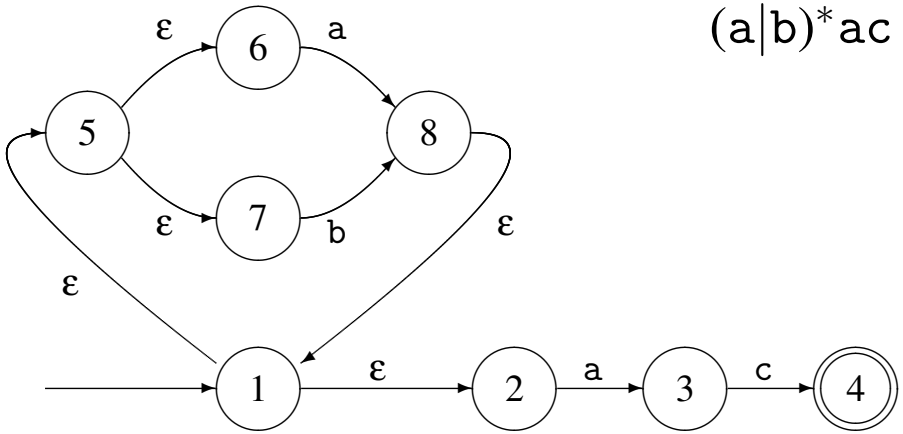
- DFA and NFA (and regular expressions) have the same expressive power
- An NFA can be converted into a DFA by the subset construction method
- Main idea: mimic the simulation of the NFA with a DFA
  - Every state of the resulting DFA corresponds to a set of states of the NFA. First state is ϵ-*closure*(s_0).
  - Transitions between states of DFA correspond to transitions between set of states in the NFA:

$$\delta(S, c) = \epsilon\text{-}closure(move(S, c))$$

  - A set of the DFA is accepting if any of the NFA states that it contains is accepting
- See INFO0016 or the reference book for more details

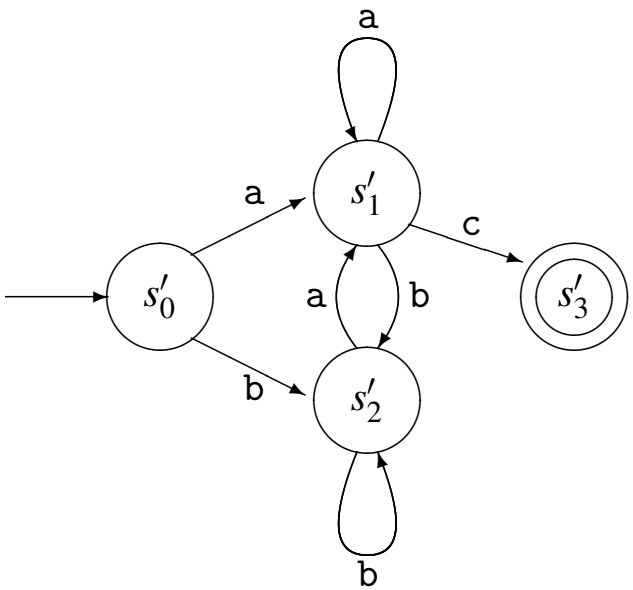
Reminder: from NFA to DFA

NFA



$(a|b)^*ac$

DFA



s'_0 $\{1, 2, 5, 6, 7\}$

s'_1 $\{3, 8, 1, 2, 5, 6, 7\}$

s'_2 $\{8, 1, 2, 5, 6, 7\}$

s'_3 $\{4\}$

(Mogensen)

Simulating a DFA

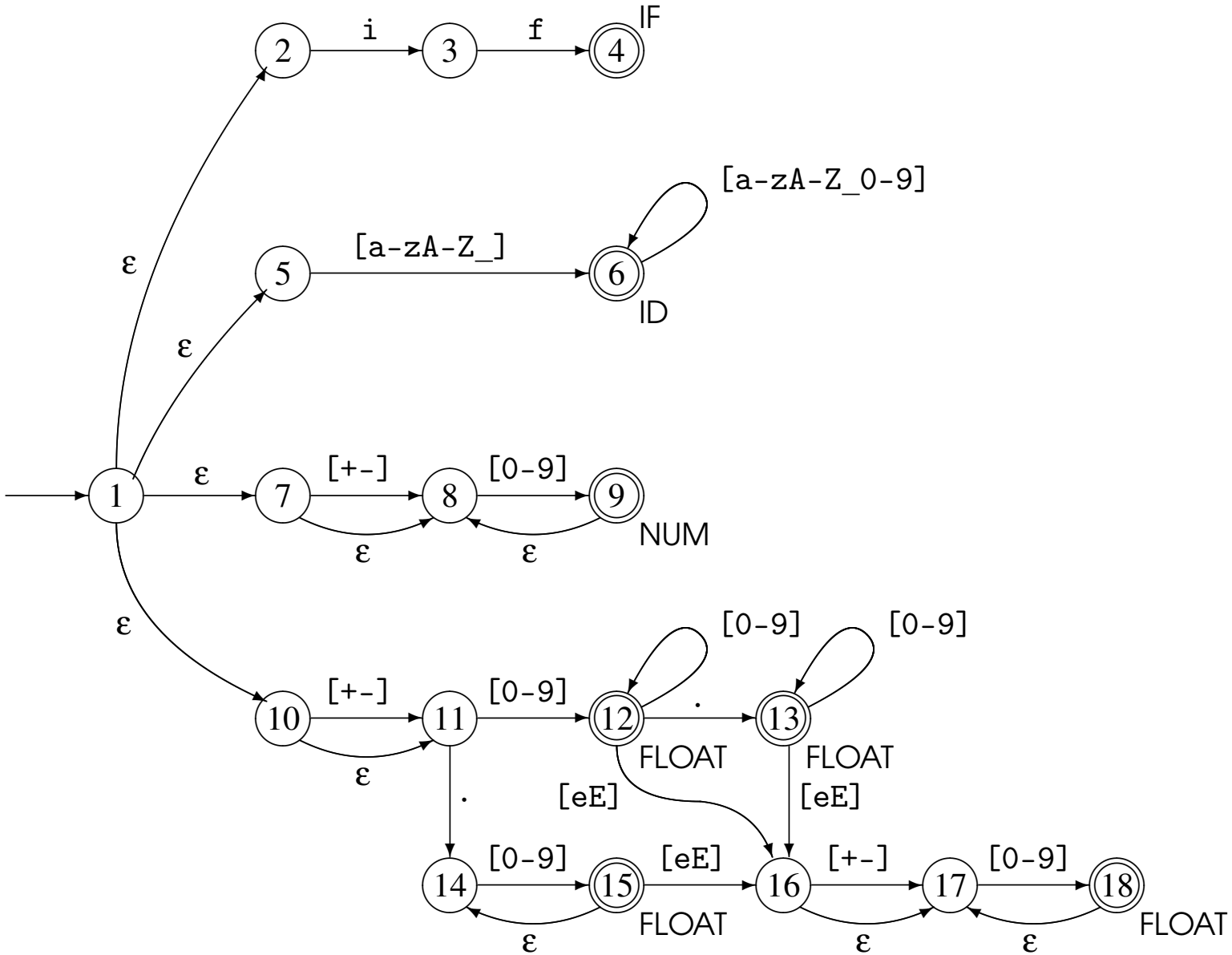
```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

- Time complexity is $O(|S|)$ for a string of length $|S|$
- Now independent of the number of states

Lexical analysis with a DFA: summary

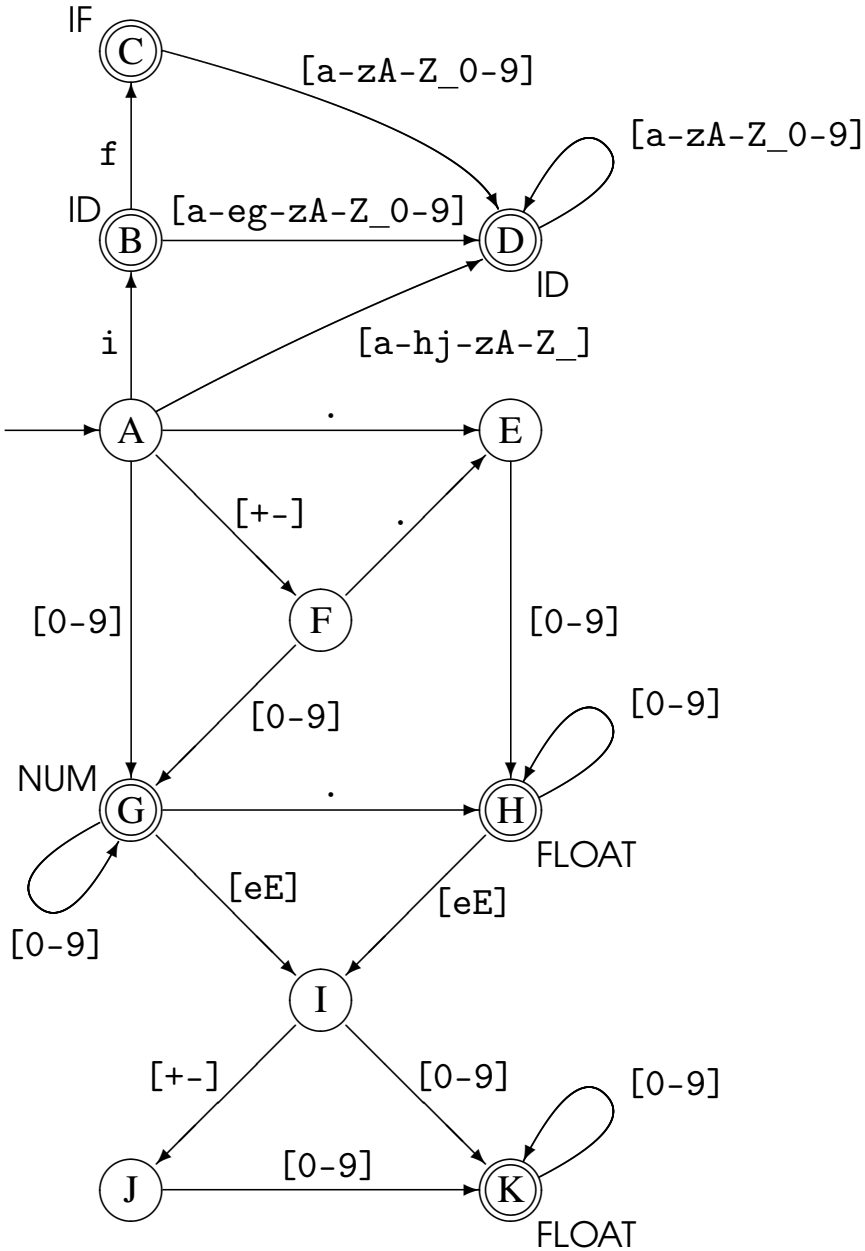
- Construct NFAs for all regular expressions
- Mark the accepting states of the NFAs by the name of the tokens they accept
- Merge them into one automaton by adding a new start state
- Convert the combined NFA to a DFA
- Convey the accepting state labeling of the NFAs to the DFA (by taking into account precedence rules)
- Scanning is done like with an NFA

Example: combined NFA for several tokens



(Mogensen)

Example: combined DFA for several tokens



Try lexing on the strings:

- *if* 17
- 3*e*-*y*

Speed versus memory

- The number of states of a DFA can grow exponentially with respect to the size of the corresponding regular expression (or NFA)
- We have to choose between low-memory and slow NFAs and high-memory and fast DFAs.

Note:

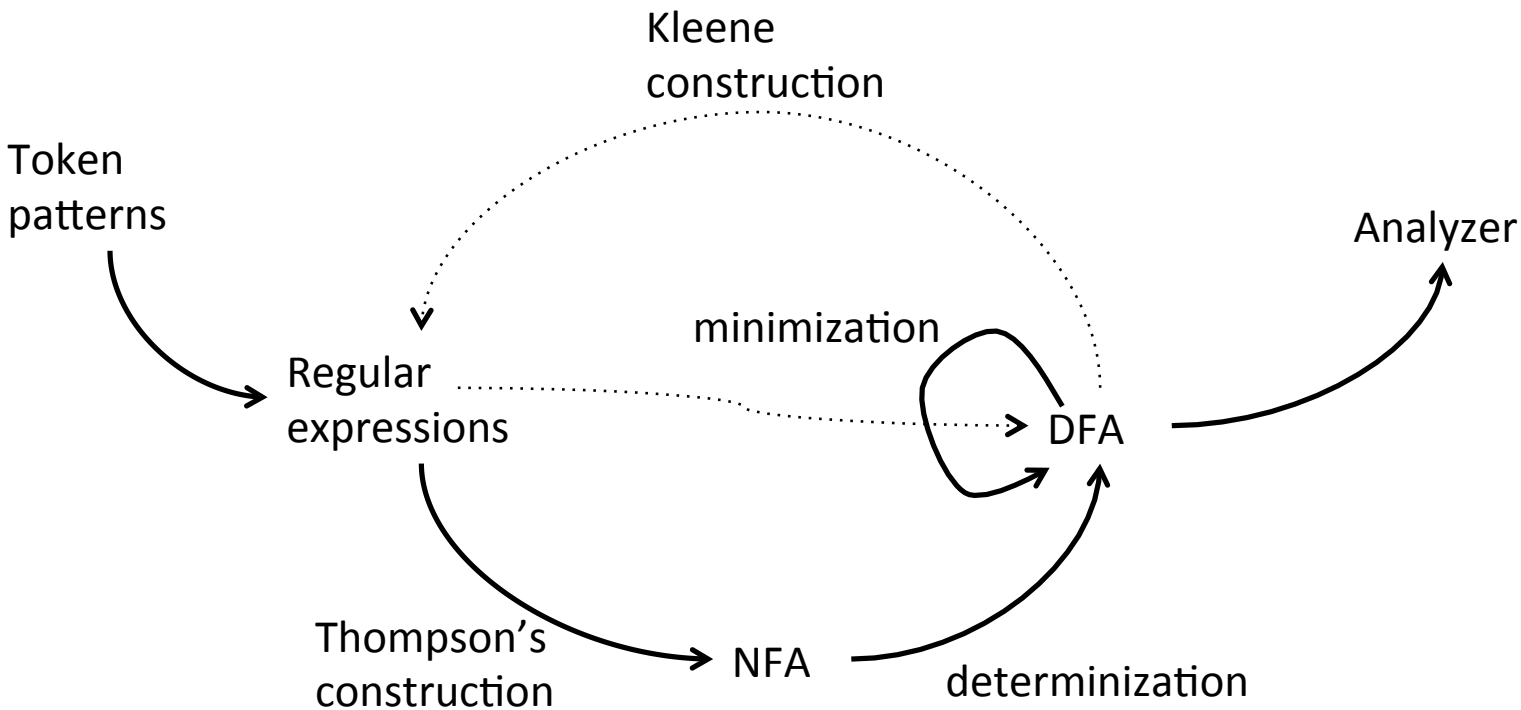
- It is possible to minimise the number of states of a DFA in $O(n \log n)$ (Hopcroft's algorithm[1])
  - Theory says that any regular language has a unique minimal DFA
  - However, the number of states may remain exponential in the size of the regular expression after minimization

[1] `http://en.wikipedia.org/wiki/DFA_minimization`

Keywords and identifiers

- Having a separate regular expression for each keyword is not very efficient.
- In practice:
  - We define only one regular expression for both keywords and identifiers
  - All keywords are stored in a (hash) table
  - Once an identifier/keyword is read, a table lookup is performed to see whether this is an identifier or a keyword
- Reduces drastically the size of the DFA
- Adding a keyword requires only to add one entry in the hash table.

Summary


Kleene
construction
Token
patterns
Analyzer
minimization
Regular
expressions
DFA
Thompson's
construction
NFA
determinization

Some langage specificities

Language specificities that make lexical analysis hard:

- Whitespaces are irrelevant in Fortran.

```
DO 5 I = 1,25
DO5I = 1.25
```

- PL/1: keywords can be used as identifiers:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

- Python block defined by indentation:

```
if w == z:
    a = b
else:
    e = f
g = h
```

(the lexical analyser needs to record current indentation and output a token for each increase/decrease in indentation)

(Keith Schwarz)

Some langage specificities

- Sometimes, nested lexical analyzers are needed
- For example, to deal with nested comments:

  ```
  /* /* where do my comments end? here? */ or here? */
  ```

  - As soon as `/*` is read, switch to another lexical analyzer that
    - only reads `/*` and `*/`,
    - counts the level of nested comments at current position (starting at 0),
    - get back to the original analyzer when it reads `*/` and the level is 0
- Other example: Javadoc (needs to interpret the comments)

NB: How could you test if your compiler accepts nested comments without generating a compilation error?

```
int nest = /*/*/0*/**/1;
```

Implementing a lexical analyzer

- In practice (and for your project), two ways:
  - Write an ad-hoc analyser
  - Use automatic tools like (F)LEX.
- First approach is more tedious. It is only useful to address specific needs.
- Second approach is more portable

Example of an ad-hoc lexical analyser

(source: http://dragonbook.stanford.edu/lecture-notes.html)

Definition of the token classes (through constants)

```
#define T_SEMICOLON  ';'      // use ASCII values for single char tokens
#define T_LPAREN  '('
#define T_RPAREN  ')'
#define T_ASSIGN  '='
#define T_DIVIDE  '/'
 ...

#define T_WHILE  257          // reserved words
#define T_IF  258
#define T_RETURN  259
 ...

#define T_IDENTIFIER  268     // identifiers, constants, etc.
#define T_INTEGER  269
#define T_DOUBLE  270
#define T_STRING  271

#define T_END  349            // code used when at end of file
#define T_UNKNOWN  350        // token was unrecognized by scanner
```

Example of an ad-hoc lexical analyser

Structure for tokens

```
struct token_t {
  int type;                    // one of the token codes from above
  union {
    char stringValue[256];     // holds lexeme value if string/identifier
    int intValue;              // holds lexeme value if integer
    double doubleValue;        // holds lexeme value if double
  } val;
};
```

Main function

```
int main(int argc, char *argv[])
{
  struct token_t token;

  InitScanner();
   while (ScanOneToken(stdin, &token) != T_END)
    ; // this is where you would process each token
  return 0;
}
```

Example of an ad-hoc lexical analyser

Initialization

```
static void InitScanner()
{
  create_reserved_table();  // table maps reserved words to token type
  insert_reserved("WHILE", T_WHILE)
  insert_reserved("IF", T_IF)
  insert_reserved("RETURN", T_RETURN)
  ....
}
```

Example of an ad-hoc lexical analyser

Scanning (single-char tokens)

```
static int ScanOneToken(FILE *fp, struct token_t *token)
{
  int i, ch, nextch;

  ch = getc(fp);     // read next char from input stream
  while (isspace(ch))     // if necessary, keep reading til non-space char
    ch = getc(fp);     // (discard any white space)

  switch(ch) {
    case '/':     // could either begin comment or T_DIVIDE op
      nextch = getc(fp);
      if (nextch == '/' || nextch == '*')
        ; // here you would skip over the comment
      else
        ungetc(nextch, fp); // fall-through to single-char token case

    case ';': case ',': case '=':  // ... and other single char tokens
      token->type = ch; // ASCII value is used as token type
      return ch;      // ASCII value used as token type
```

Example of an ad-hoc lexical analyser

Scanning: keywords

```
case 'A': case 'B': case 'C':   // ... and other upper letters
  token->val.stringValue[0] = ch;
  for (i = 1; isupper(ch = getc(fp)); i++) // gather uppercase
    token->val.stringValue[i] = ch;
  ungetc(ch, fp);
  token->val.stringValue[i] = '\0';  // lookup reserved word
  token->type = lookup_reserved(token->val.stringValue);
  return token->type;
```

Scanning: identifier

```
case 'a': case 'b': case 'c':  // ... and other lower letters
  token->type = T_IDENTIFIER;
  token->val.stringValue[0] = ch;
  for (i = 1; islower(ch = getc(fp)); i++)
    token->val.stringValue[i] = ch; // gather lowercase
  ungetc(ch, fp);
  token->val.stringValue[i] = '\0';
  if (lookup_symtab(token->val.stringValue) == NULL)
    add_symtab(token->val.stringValue); // get symbol for ident
  return T_IDENTIFIER;
```

Example of an ad-hoc lexical analyser

Scanning: number

```
case '0': case '1': case '2': case '3':  //....  and other digits
  token->type = T_INTEGER;
  token->val.intValue = ch - '0';
  while (isdigit(ch = getc(fp))) // convert digit char to number
    token->val.intValue = token->val.intValue * 10 + ch - '0';
  ungetc(ch, fp);
  return T_INTEGER;
```

Scanning: EOF and default

```
case EOF:
  return T_END;

default:    // anything else is not recognized
  token->val.intValue = ch;
  token->type = T_UNKNOWN;
  return T_UNKNOWN;
```

Flex

- `flex` is a free implementation of the Unix `lex` program
- `flex` implements what we have seen:
  - It takes regular expressions as input
  - It generates a combined NFA
  - It converts it to an equivalent DFA
  - It minimizes the automaton as much as possible
  - It generates C code that implements it
  - It handles conflicts with the longest matching prefix principle and a preference order on the tokens.
- More information
  - `http://flex.sourceforge.net/manual/`

Input file

- Input files are structured as follows:

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

- Declarations and User subroutines are copied without modifications to the generated C file.
- Definitions specify options and name definitions (to simplify the rules)
- Rules: specify the patterns for the tokens to be recognized

Rules

- In the form:

```
pattern1 action1
pattern2 action2
...
```

- Patterns are defined as regular expressions. Actions are blocks of C code.
- When a sequence is read that matches the pattern, the C code of the action is executed
- Examples:

```
[0-9]+ {printf("This is a number");}
[a-z]+ {printf("This is symbol");}
```

Regular expressions

- Many shortcut notations are permitted in regular expressions:
  - `[]`, `-`, `+`, `*`, ?: as defined previously
  - `.`: a dot matches any character (except newline)
  - `[^x]`: matches the complement of the set of characters in `x` (ex: all non-digit characters `[^0-9]`).
  - `x{n,m}`: `x` repeated between `n` and `m` times
  - `"x"`: matches `x` even if `x` contains special characters (ex: `"x*"` matches `x` followed by a star).
  - `{name}`: replace with the pattern defined earlier in the definition section of the input file

Interacting with the scanner

- User subroutines and action may interact with the generated scanner through global variables:
  - `yylex`: scan tokens from the global input file `yyin` (defaults to stdin). Continues until it reaches the end of the file or one of its actions executes a return statement.
  - `yytext`: a null-terminated string (of length `yyleng`) containing the text of the lexeme just recognized.
  - `yylval`: store the attributes of the token
  - `yylloc`: location of the tokens in the input file (line and column)
  - ...

Example 1: hiding numbers

- hide-digits.l:

```
%%
[0-9]+ printf("?");
. ECHO;
```

- To build and run the program:

```
% flex hide-digits.l
% gcc -o hide-digits lex.yy.c -ll
% ./hide-digits
```

Example 2: wc

- count.l:

```
%{
   int numChars = 0, numWords = 0, numLines = 0;
%}
%%
\n           {numLines++; numChars++;}
[^ \t\n]+    {numWords++; numChars += yyleng;}
.            {numChars++;}
%%

int main() {
    yylex();
    printf("%d\t%d\t%d\n", numChars, numWords, numLines);
}
```

- To build and run the program:

```
% flex count.l
% gcc -o count lex.yy.c -ll
% ./count < count.l
```

Example 3: typical compiler

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}
```

Example 3: typical compiler

User defined subroutines

```
%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

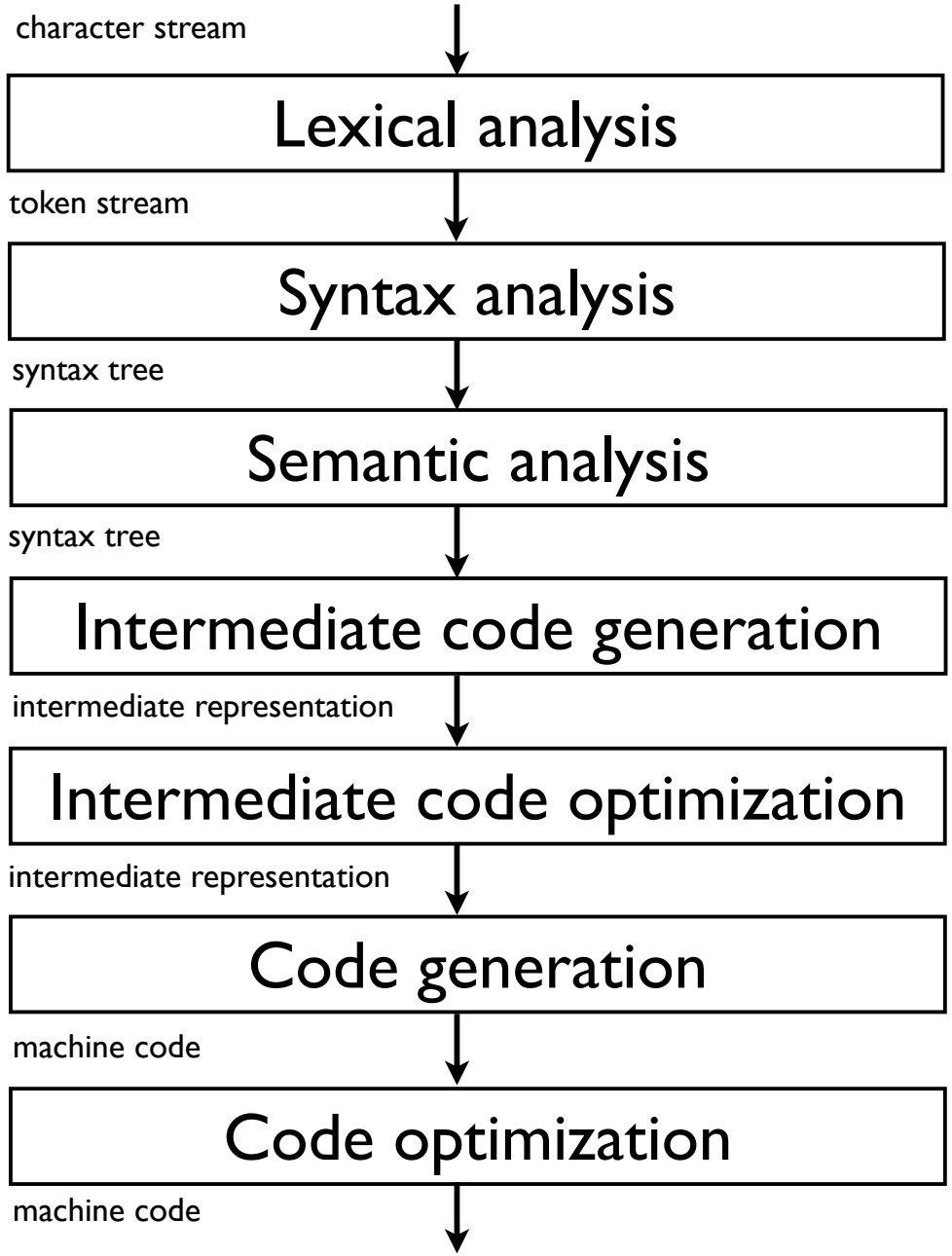
Part 3

Syntax analysis

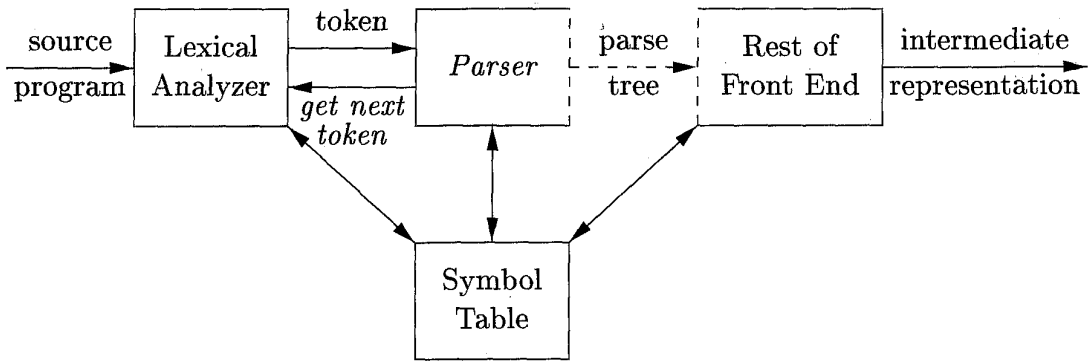
Outline

1. Introduction
2. Context-free grammar
3. Top-down parsing
4. Bottom-up parsing
5. Conclusion and some practical considerations

Structure of a compiler

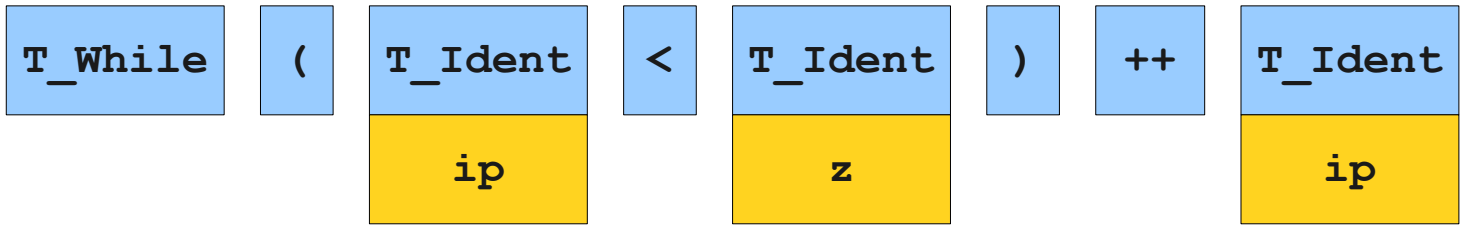


Syntax analysis



- Goals:
  - recombine the tokens provided by the lexical analysis into a structure (called *a syntax tree*)
  - Reject invalid texts by reporting *syntax errors*.
- Like lexical analysis, syntax analysis is based on
  - the definition of valid programs based on some formal languages,
  - the derivation of an algorithm to detect valid words (programs) from this language
- Formal language: context-free grammars
- Two main algorithm families: Top-down parsing and Bottom-up parsing

Example

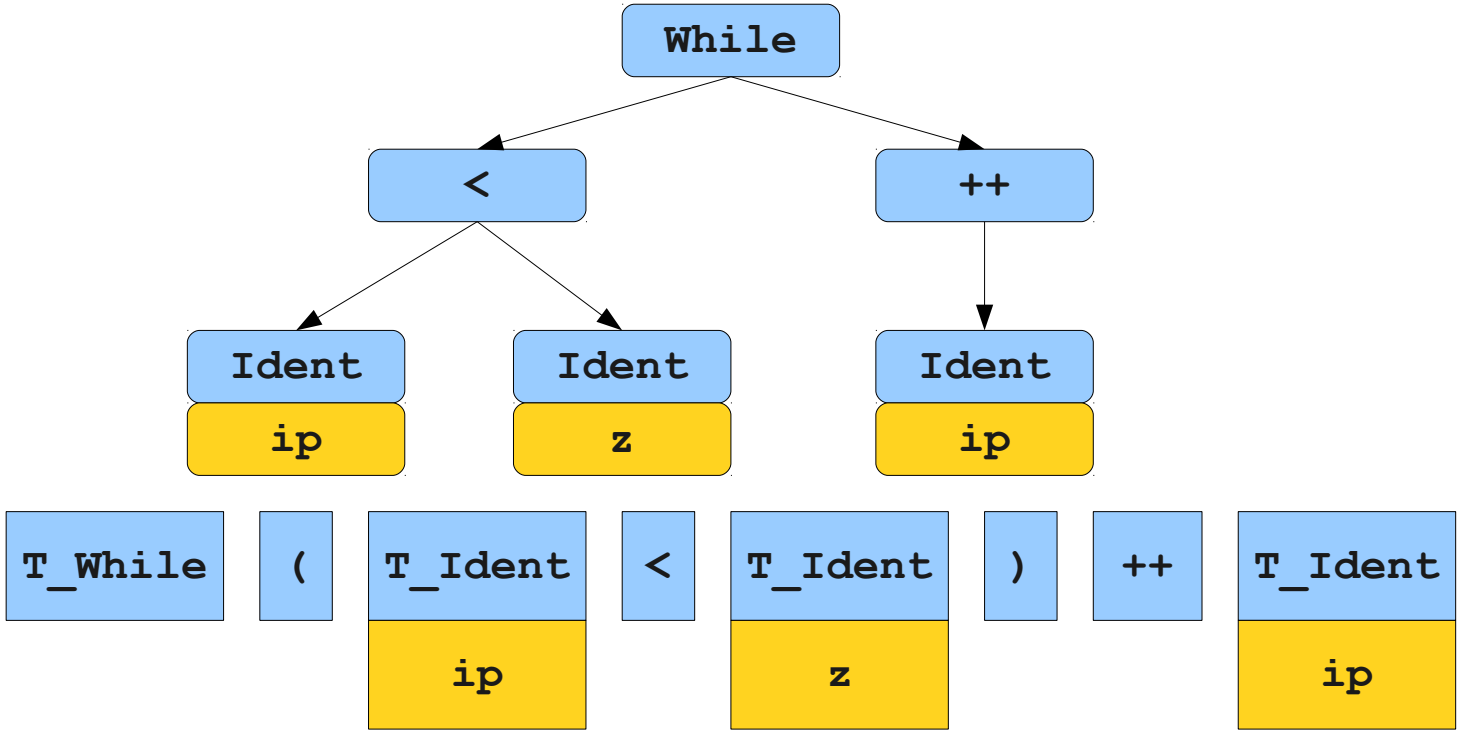


w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Example



w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Reminder: grammar

- A grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where:
  - V is an alphabet,
  - $\Sigma \subseteq V$ is the set of terminal symbols ($V - \Sigma$ is the set of nonterminal symbols),
  - $R \subseteq (V^+ \times V^*)$ is a finite set of production rules
  - $S \in V - \Sigma$ is the start symbol.
- Notations:
  - Nonterminal symbols are represented by uppercase letters: A,B,...
  - Terminal symbols are represented by lowercase letters: a,b,...
  - Start symbol written as S
  - Empty word: ϵ
  - A rule $(\alpha, \beta) \in R : \alpha \to \beta$
  - Rule combination: $A \to \alpha|\beta$
- Example: $\Sigma = \{a, b, c\}$, $V - \Sigma = \{S, R\}$, $R =$

$$\begin{aligned} S &\to R \\ S &\to aSc \\ R &\to \epsilon \\ R &\to RbR \end{aligned}$$

Reminder: derivation and language

Definitions:

- v can be *derived in one step* from u by G (noted $v \Rightarrow u$) iff $u = xu'y$, $v = xv'y$, and $u' \rightarrow v'$
- v can be *derived in several steps* from u by G (noted $v \overset{*}{\Rightarrow} u$) iff $\exists k \geq 0$ and $v_0 \dots v_k \in V^+$ such that $u = v_0$, $v = v_k$, $v_i \Rightarrow v_{i+1}$ for $0 \leq i < k$
- The *language generated by a grammar* G is the set of words that can be derived from the start symbol:

$$L = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$$

Example: derivation of *aabcc* from the previous grammar

$$\underline{S} \Rightarrow a\underline{S}c \Rightarrow aa\underline{S}cc \Rightarrow aa\underline{R}cc \Rightarrow aa\underline{R}bRcc \Rightarrow aab\underline{R}cc \Rightarrow aabcc$$

Reminder: type of grammars

Chomsky's grammar hierarchy:

- Type 0: free or unrestricted grammars
- Type 1: context sensitive grammars
  - productions of the form $uXw \rightarrow uvw$, where u, v, w are arbitrary strings of symbols in V, with v non-null, and X a single nonterminal
- Type 2: context-free grammars (CFG)
  - productions of the form $X \rightarrow v$ where v is an arbitrary string of symbols in V, and X a single nonterminal.
- Type 3: regular grammars
  - Productions of the form $X \rightarrow a$, $X \rightarrow aY$ or $X \rightarrow \epsilon$ where X and Y are nonterminals and a is a terminal (equivalent to regular expressions and finite state automata)

Context-free grammars

- Regular languages are too limited for representing programming languages.
- Examples of languages not representable by a regular expression:
  - $L = \{a^n b^n | n \geq 0\}$
  - Balanced parentheses
    $L = \{\epsilon, (), (()), ()(), ((())), (())() \ldots\}$
  - Scheme programs
    $L = \{1, 2, 3, \ldots, (lambda(x)(+x1))\}$
- Context-free grammars are typically used for describing programming language syntaxes.
  - They are sufficient for most languages
  - They lead to efficient parsing algorithms

Context-free grammars for programming languages

- Terminals of the grammars are typically the tokens derived by the lexical analysis (in bold in rules)
- Divide the language into several syntactic categories (sub-languages)
- Common syntactic categories
  - Expressions: calculation of values
  - Statements: express actions that occur in a particular sequence
  - Declarations: express properties of names used in other parts of the program

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} - \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} * \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} / \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \textbf{id} \\
\mathit{Exp} & \rightarrow & (\mathit{Exp})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \textbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Stat}; \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ Else } \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat}
\end{array}
$$

Derivation for context-free grammar

- Like for a general grammar
- Because there is only one nonterminal in the LHS of each rule, their order of application does not matter
- Two particular derivations
  - left-most: always expand first the left-most nonterminal (important for parsing)
  - right-most: always expand first the right-most nonterminal (canonical derivation)
- Examples

$$S \rightarrow aTb|c$$
$$T \rightarrow cSS|S$$

$w = accacbb$

Left-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow accSb \Rightarrow$
$accaTbb \Rightarrow accaSbb \Rightarrow accacbb$

Right-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow acSaTbb \Rightarrow$
$acSaSbb \Rightarrow acSacbb \Rightarrow accacbb$

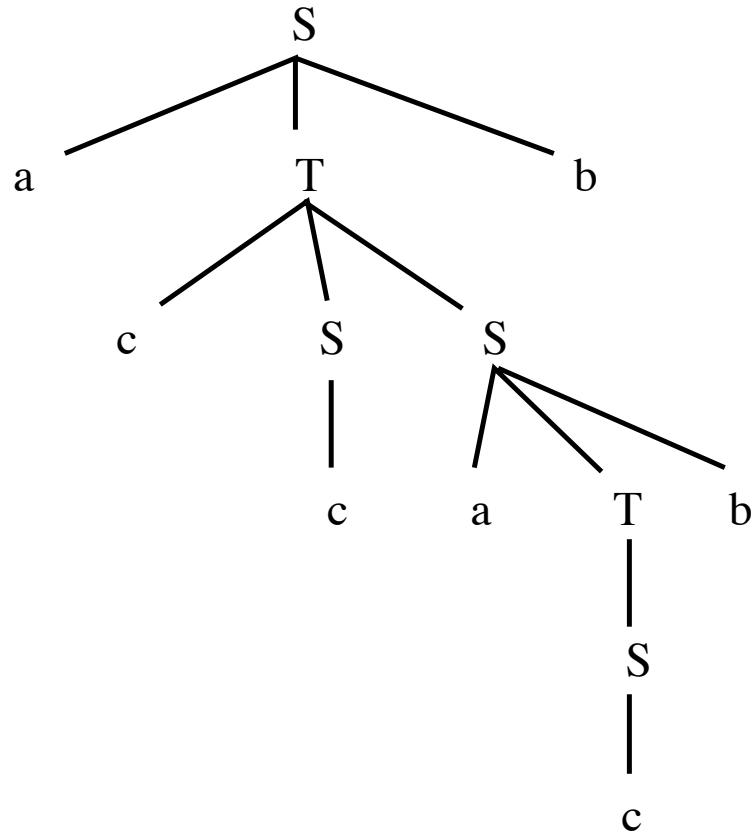
Parse tree

- A parse tree abstracts the order of application of the rules
  - Each interior node represents the application of a production
  - For a rule $A \to X_1 X_2 \dots X_k$, the interior node is labeled by A and the children from left to right by $X_1, X_2, \dots, X_k$.
  - Leaves are labeled by nonterminals or terminals and read from left to right represent a string generated by the grammar
- A derivation encodes how to produce the input
- A parse tree encodes the structure of the input
- Syntax analysis = recovering the parse tree from the tokens

Parse trees

$$S \rightarrow aTb|c$$
$$T \rightarrow cSS|S$$

$$w = accacbb$$

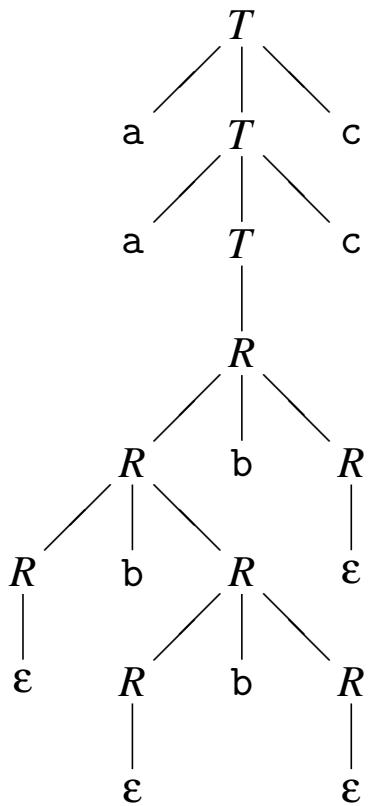


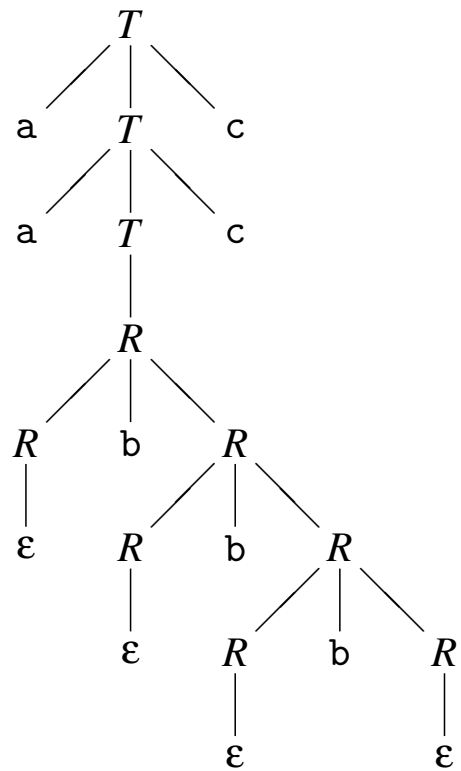
Left-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow accSb \Rightarrow$
$accaTbb \Rightarrow accaSbb \Rightarrow accacbb$

Right-most derivation:
$S \Rightarrow aTb \Rightarrow acSSb \Rightarrow acSaTbb \Rightarrow$
$acSaSbb \Rightarrow acSacbb \Rightarrow accacbb$

Parse tree

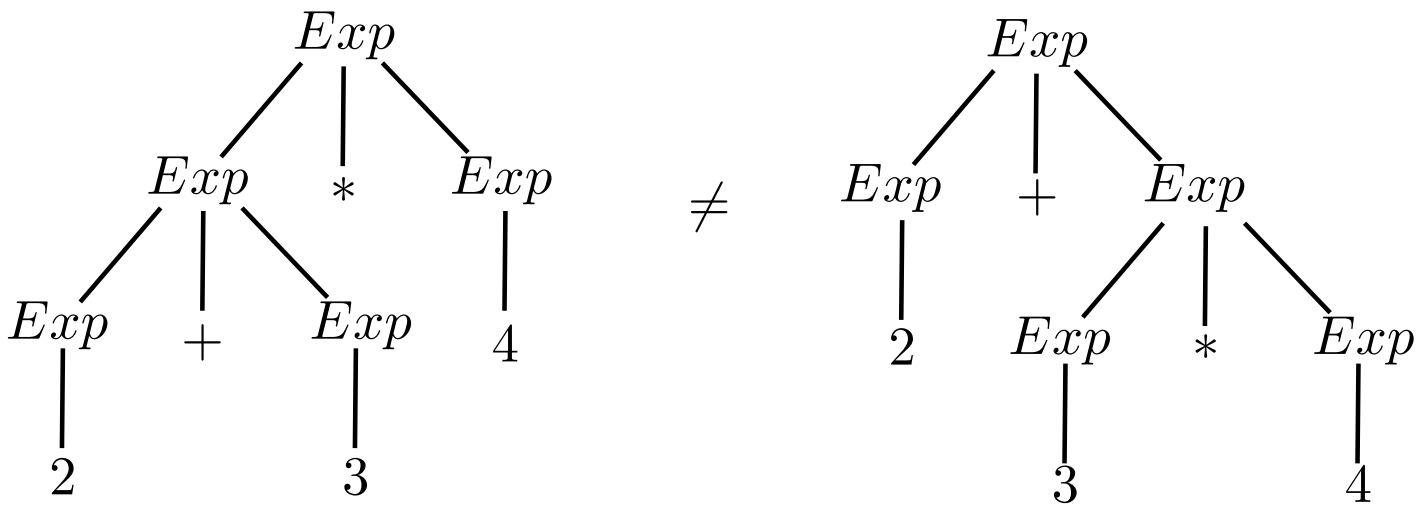
$$
\begin{aligned}
T &\rightarrow R \\
T &\rightarrow aTc \\
R &\rightarrow \epsilon \\
R &\rightarrow RbR
\end{aligned}
$$





Ambiguity

- The order of derivation does not matter but the chosen production rules do
- **Definition:** A CFG is ambiguous if there is at least one string with two or more parse trees
- Ambiguity is not problematic when dealing with flat strings. It is when dealing with language semantics



Detecting and solving Ambiguity

- There is no mechanical way to determine if a grammar is (un)ambiguous (this is an undecidable problem)
- In most practical cases however, it is easy to detect and prove ambiguity.
  E.g., any grammar containing $N \to N\alpha N$ is ambiguous (two parse trees for $N\alpha N\alpha N$).
- How to deal with ambiguities?
  - Modify the grammar to make it unambiguous
  - Handle these ambiguities in the parsing algorithm
- Two common sources of ambiguity in programming languages
  - Expression syntax (operator precedences)
  - Dangling else

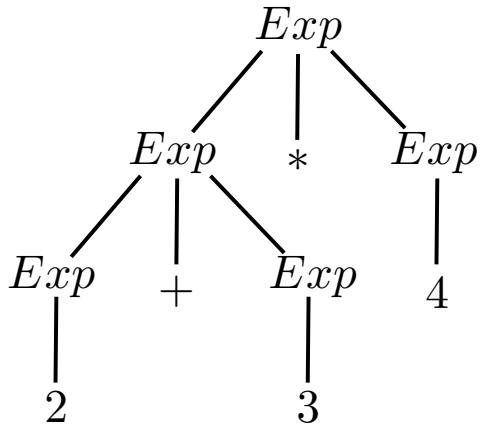
Operator precedence

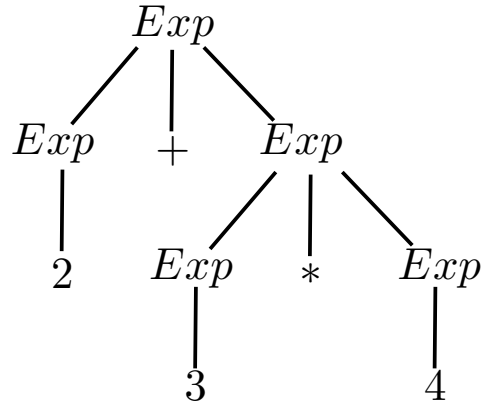
- This expression grammar is ambiguous

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Exp \\
Exp & \rightarrow & Exp - Exp \\
Exp & \rightarrow & Exp * Exp \\
Exp & \rightarrow & Exp / Exp \\
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & (Exp)
\end{array}
$$

(it contains $N \rightarrow N\alpha N$)

- Parsing of $2 + 3 * 4$





Operator associativity

- Types of operator associativity:
  - An operator $\oplus$ is left-associative if $a \oplus b \oplus c$ must be evaluated from left to right, i.e., as $(a \oplus b) \oplus c$
  - An operator $\oplus$ is right-associative if $a \oplus b \oplus c$ must be evaluated from right to left, i.e., as $a \oplus (b \oplus c)$
  - An operator $\oplus$ is non-associative if expressions of the form $a \oplus b \oplus c$ are not allowed
- Examples:
  - $-$ and $/$ are typically left-associative
  - $+$ and $*$ are mathematically associative (left or right). By convention, we take them left-associative as well
  - List construction in functional languages is right-associative
  - Arrows operator in C is right-associative (`a->b->c` is equivalent to `a->(b->c)`)
  - In Pascal, comparison operators are non-associative (you can not write $2 < 3 < 4$)

Rewriting ambiguous expression grammars

- Let's consider the following ambiguous grammar:

$$\begin{aligned} E &\rightarrow E \oplus E \\ E &\rightarrow \text{num} \end{aligned}$$

- If $\oplus$ is left-associative, we rewrite it as a left-recursive (a recursive reference only to the left). If $\oplus$ is right-associative, we rewrite it as a right-recursive (a recursive reference only to the right).

$\oplus$ left-associative

$$\begin{aligned} E &\rightarrow E \oplus E' \\ E &\rightarrow E' \\ E' &\rightarrow \text{num} \end{aligned}$$

$\oplus$ right-associative

$$\begin{aligned} E &\rightarrow E' \oplus E \\ E &\rightarrow E' \\ E' &\rightarrow \text{num} \end{aligned}$$

Mixing operators of different precedence levels

- Introduce a different nonterminal for each precedence level

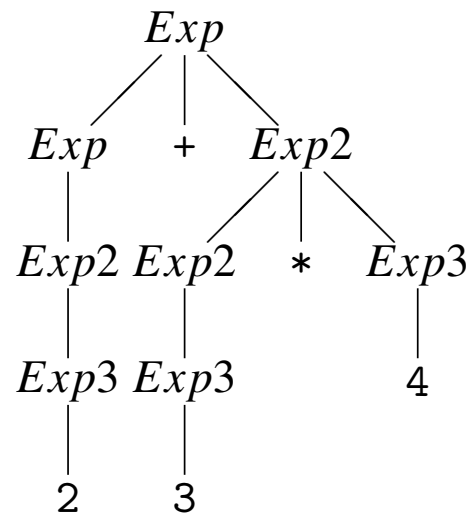
Ambiguous

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Exp \\
Exp & \rightarrow & Exp - Exp \\
Exp & \rightarrow & Exp * Exp \\
Exp & \rightarrow & Exp / Exp \\
Exp & \rightarrow & \text{num} \\
Exp & \rightarrow & (Exp)
\end{array}
$$

Non-ambiguous

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \text{num} \\
Exp3 & \rightarrow & (Exp)
\end{array}
$$

Parse tree for $2 + 3 * 4$



Dangling else

- Else part of a condition is typically optional

$$\begin{aligned} \mathit{Stat} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ Else } \mathit{Stat} \\ \mathit{Stat} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \end{aligned}$$

- How to match `if p then if q then s1 else s2`?
- Convention: `else` matches the closest not previously matched `if`.
- Unambiguous grammar:

$$\begin{aligned} \mathit{Stat} &\rightarrow \mathit{Matched} \mid \mathit{Unmatched} \\ \mathit{Matched} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Matched} \textbf{ else } \mathit{Matched} \\ \mathit{Matched} &\rightarrow \text{"Any other statement"} \\ \mathit{Unmatched} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \\ \mathit{Unmatched} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Matched} \textbf{ else } \mathit{Unmatched} \end{aligned}$$

End-of-file marker

- Parsers must read not only terminal symbols such as $+, -$, **num** , but also the end-of-file
- We typically use \$ to represent end of file
- If S is the start symbol of the grammar, then a new start symbol S' is added with the following rules $S' \rightarrow S\$$.

$$
\begin{array}{rcl}
S & \rightarrow & Exp\$ \\
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \text{num} \\
Exp3 & \rightarrow & (Exp)
\end{array}
$$

Non-context free languages

- Some syntactic constructs from typical programming languages cannot be specified with CFG
- Example 1: ensuring that a variable is declared before its use
  - $L_1 = \{wcw | w \text{ is in } (a|b)^*\}$ is not context-free
  - In C and Java, there is one token for all identifiers
- Example 2: checking that a function is called with the right number of arguments
  - $L_2 = \{a^n b^m c^n d^m | n \geq 1 \text{ and } m \geq 1\}$ is not context-free
  - In C, the grammar does not count the number of function arguments

$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & \textbf{id}\ (\mathit{expr\_list}) \\ \mathit{expr\_list} & \rightarrow & \mathit{expr\_list}, \mathit{expr} \\ & | & \mathit{expr} \end{array}$$

- These constructs are typically dealt with during semantic analysis

Backus-Naur Form

- A text format for describing context-free languages
- We ask you to provide the source grammar for your project in this format
- Example:

```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <factor> "*" <term>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
<digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

- More information:
  `http://en.wikipedia.org/wiki/Backus-Naur_form`

Outline

1. Introduction

2. Context-free grammar

3. Top-down parsing

4. Bottom-up parsing

5. Conclusion and some practical considerations

Syntax analysis

- Goals:
  - Checking that a program is accepted by the context-free grammar
  - Building the parse tree
  - Reporting syntax errors
- Two ways:
  - Top-down: from the start symbol to the word
  - Bottom-up: from the word to the start symbol

Top-down and bottom-up: example

Grammar:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA|\epsilon \\ B &\rightarrow b|bB \end{aligned}$$

Top-down parsing of *aaab*

S	
AB	$S \rightarrow AB$
aAB	$A \rightarrow aA$
$aaAB$	$A \rightarrow aA$
$aaa\epsilon B$	$A \rightarrow \epsilon$
$aaab$	$B \rightarrow b$

Bottom-up parsing of *aaab*

$aaab$	
$aaa\epsilon b$	(insert ϵ)
$aaaAb$	$A \rightarrow \epsilon$
$aaAb$	$A \rightarrow aA$
aAb	$A \rightarrow aA$
Ab	$A \rightarrow aA$
AB	$B \rightarrow b$
S	$S \rightarrow AB$

A naive top-down parser

- A very naive parsing algorithm:
  - Generate all possible parse trees until you get one that matches your input
  - To generate all parse trees:
    1. Start with the root of the parse tree (the start symbol of the grammar)
    2. Choose a non-terminal A at one leaf of the current parse tree
    3. Choose a production having that non-terminal as LHS, eg., $A \rightarrow X_1 X_2 \ldots X_k$
    4. Expand the tree by making X_1,X_2,. . . ,X_k, the children of A.
    5. Repeat at step 2 until all leaves are terminals
    6. Repeat the whole procedure by changing the productions chosen at step 3

  (Note: the choice of the non-terminal in Step 2 is irrevelant for a context-free grammar)
- This algorithm is very inefficient, does not always terminate, etc.

Top-down parsing with backtracking

- Modifications of the previous algorithm:
  1. Depth-first development of the parse tree (corresponding to a left-most derivation)
  2. Process the terminals in the RHS during the development of the tree, checking that they match the input
  3. If they don't at some step, stop expansion and restart at the previous non-terminal with another production rules (backtracking)
- Depth-first can be implemented by storing the unprocessed symbols on a stack
- Because of the left-most derivation, the inputs can be processed from left to right

Backtracking example

$$
\begin{aligned}
S &\rightarrow bab \\
S &\rightarrow bA \\
A &\rightarrow d \\
A &\rightarrow cA
\end{aligned}
$$

$w = bcd$

Stack	Inputs	Action
S	bcd	Try $S \rightarrow bab$
bab	bcd	match b
ab	cd	dead-end, backtrack
S	bcd	Try $S \rightarrow bA$
bA	bcd	match b
A	cd	Try $A \rightarrow d$
d	cd	dead-end, backtrack
A	cd	Try $A \rightarrow cA$
cA	cd	match c
A	d	Try $A \rightarrow d$
d	d	match d
		Success!

Top-down parsing with backtracking

- General algorithm (to match a word w):

```
Create a stack with the start symbol
X = POP()
a = GETNEXTTOKEN()
while (True)
    if (X is a nonterminal)
        Pick next rule to expand X → Y1 Y2 ... Yk
        Push Yk, Yk−1, ..., Y1 on the stack
        X = POP()
    elseif (X == $ and a == $)
        Accept the input
    elseif (X == a)
        a = GETNEXTTOKEN()
        X = POP()
    else
        Backtrack
```

- Ok for small grammars but still untractable and very slow for large grammars
- Worst-case exponential time in case of syntax error

Another example

$$
\begin{array}{lcl}
S & \to & aSbT \\
S & \to & cT \\
S & \to & d \\
T & \to & aT \\
T & \to & bS \\
T & \to & c
\end{array}
$$

$w = accbbadbc$

Stack	Inputs	Action
S	$accbbadbc$	Try $S \to aSbT$
$aSbT$	$accbbadbc$	match a
SbT	$accbbadbc$	Try $S \to aSbT$
$aSbTbT$	$accbbadbc$	match a
$SbTbT$	$ccbbadbc$	Try $S \to cT$
$cTbTbT$	$ccbbadbc$	match c
$TbTbT$	$cbbadbc$	Try $T \to c$
$cbTbT$	$cbbadbc$	match cb
TbT	$badbc$	Try $T \to bS$
$bSbT$	$badbc$	match b
SbT	$adbc$	Try $S \to aSbT$
$aSbT$	$adbc$	match a
$\ldots$	$\ldots$	$\ldots$
c	c	match c
		Success!

Predictive parsing

- Predictive parser:
  - In the previous example, the production rule to apply can be predicted based solely on the next input symbol and the current nonterminal
  - Much faster than backtracking but this trick works only for some specific grammars
- Grammars for which top-down predictive parsing is possible by looking at the next symbol are called $LL(1)$ grammars:
  - L: left-to-right scan of the tokens
  - L: leftmost derivation
  - (1): One token of lookahead
- Predicted rules are stored in a parsing table M:
  - $M[X, a]$ stores the rule to apply when the nonterminal X is on the stack and the next input terminal is a

Example: parse table

S → E$
E → int
E → (E Op E)
Op → +
Op → *

	int	(	)	+	*	$
S	E$	E$				
E	int	(E Op E)				
Op				+	*	

(Keith Schwarz)

Example: successfull parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → -

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$
$	$

(Keith Schwarz)

Example: erroneous parsing

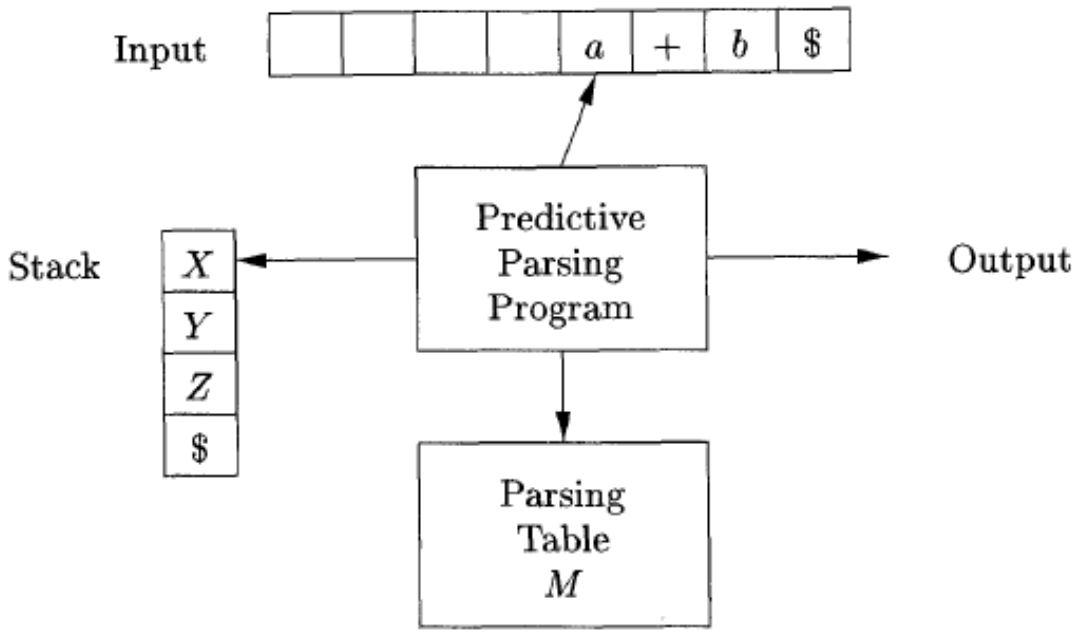
1. S → E$
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

S	**(int (int))$**
E$	**(int (int))$**
(E Op E**)$**	**(int (int))$**
E Op E**)$**	**int (int))$**
int Op E**)$**	**int (int))$**
Op E**)$**	**(int))$**

(Keith Schwarz)

Table-driven predictive parser



(Dragonbook)

Table-driven predictive parser

Create a stack with the start symbol
X = POP()
a = GETNEXTTOKEN()
while (True)
    if (X is a nonterminal)
        if ($M[X, a] == NULL$)
            Error
        elseif ($M[X, a] == X \rightarrow Y_1 Y_2 \ldots Y_k$)
            Push $Y_k, Y_{k-1}, \ldots, Y_1$ on the stack
            X = POP()
    elseif ($X ==$ \$ and $a ==$ \$)
        Accept the input
    elseif ($X == a$)
        a = GETNEXTTOKEN()
        X = POP()
    else
        Error

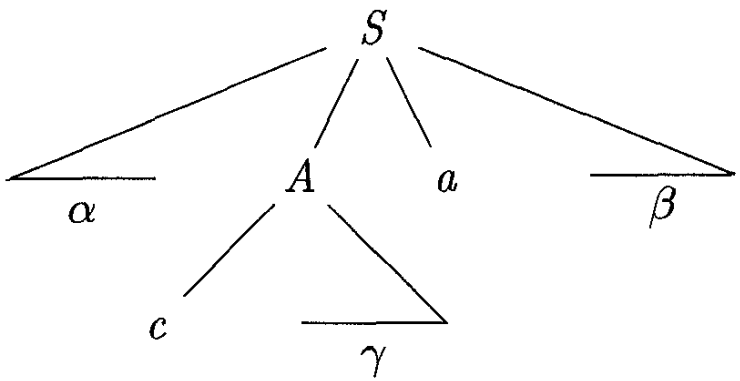
$LL(1)$ grammars and parsing

Three questions we need to address:

- How to build the table for a given grammar?
- How to know if a grammar is $LL(1)$?
- How to change a grammar to make it $LL(1)$?

Building the table

- It is useful to define three functions
  (with A a nonterminal and α any sequence of grammar symbols):
  - *Nullable*(α) is true if $\alpha \stackrel{*}{\Rightarrow} \epsilon$
  - *First*(α) returns the set of terminals c such that $\alpha \stackrel{*}{\Rightarrow} c\gamma$ for some (possibly empty) sequence γ of grammar symbols
  - *Follow*(A) returns the set of terminals a such that $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, where α and β are (possibly empty) sequences of grammar symbols



($c \in$ *First*(A) and $a \in$ *Follow*(A))

Building the table from *First*, *Follow*, and *Nullable*

To construct the table:

- Start with the empty table
- For each production $A \to \alpha$:
  - add $A \to \alpha$ to $M[A, a]$ for each terminal a in $First(\alpha)$
  - If $Nullable(\alpha)$, add $A \to \alpha$ to $M[A, a]$ for each a in $Follow(A)$

First rule is obvious. Illustration of the second rule:

$$\begin{array}{lcl} S & \to & Ab \\ A & \to & c \\ A & \to & \epsilon \end{array} \qquad \begin{array}{rcl} Nullable(A) & = & True \\ First(A) & = & \{c\} \\ Follow(A) & = & \{b\} \end{array} \qquad M[A, b] = A \to \epsilon$$

$LL(1)$ grammars

- Three situations:
  - $M[A, a]$ is empty: no production is appropriate. We can not parse the sentence and have to report a syntax error
  - $M[A, a]$ contains one entry: perfect !
  - $M[A, a]$ contains two entries: the grammar is not appropriate for predictive parsing (with one token lookahead)
- **Definition:** A grammar is $LL(1)$ if its parsing table contains at most one entry in each cell or, equivalently, if for all production pairs $A \rightarrow \alpha|\beta$
  - $First(\alpha) \cap First(\beta) = \emptyset$,
  - $Nullable(\alpha)$ and $Nullable(\beta)$ are not both true,
  - if $Nullable(\beta)$, then $First(\alpha) \cap Follow(A) = \emptyset$
- Example of a non $LL(1)$ grammar:

$$\begin{aligned} S &\rightarrow Ab \\ A &\rightarrow b \\ A &\rightarrow \epsilon \end{aligned}$$

Computing *Nullable*

Algorithm to compute *Nullable* for all grammar symbols

Initialize *Nullable* to *False*.
repeat
    for each production $X \to Y_1 Y_2 \ldots Y_k$
        if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
            $\mathit{Nullable}(X) = \mathit{True}$
until *Nullable* did not change in this iteration.

Algorithm to compute *Nullable* for any string $\alpha = X_1 X_2 \ldots X_k$:

if ($X_1 \ldots X_k$ are all nullable)
    $\mathit{Nullable}(\alpha) = \mathit{True}$
else
    $\mathit{Nullable}(\alpha) = \mathit{False}$

Computing *First*

Algorithm to compute *First* for all grammar symbols

Initialize *First* to empty sets. **for** each terminal Z
    $First(Z) = \{Z\}$
repeat
    for each production $X \to Y_1 Y_2 \ldots Y_k$
        for $i = 1$ **to** k
            if $Y_1 \ldots Y_{i-1}$ are all nullable (or $i = 1$)
                $First(X) = First(X) \cup First(Y_i)$
until *First* did not change in this iteration.

Algorithm to compute *First* for any string $\alpha = X_1 X_2 \ldots X_k$:

Initialize $First(\alpha) = \emptyset$
for $i = 1$ **to** k
    if $X_1 \ldots X_{i-1}$ are all nullable (or $i = 1$)
        $First(\alpha) = First(\alpha) \cup First(X_i)$

Computing *Follow*

To compute *Follow* for all nonterminal symbols

Initialize *Follow* to empty sets.
repeat
  for each production $X \to Y_1 Y_2 \ldots Y_k$
    for $i = 1$ **to** k, **for** $j = i + 1$ **to** k
      if $Y_{i+1} \ldots Y_k$ are all nullable (or $i = k$)
        $Follow(Y_i) = Follow(Y_i) \cup Follow(X)$
      if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or $i + 1 = j$)
        $Follow(Y_i) = Follow(Y_i) \cup First(Y_j)$
until *Follow* did not change in this iteration.

Example

Compute the parsing table for the following grammar:

$$
\begin{array}{rcl}
S & \rightarrow & E\$ \\
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \\
E' & \rightarrow & -TE' \\
E' & \rightarrow & \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \\
T' & \rightarrow & /FT' \\
T' & \rightarrow & \epsilon \\
F & \rightarrow & \textbf{id} \\
F & \rightarrow & \textbf{num} \\
F & \rightarrow & (E)
\end{array}
$$

Example

Nonterminals	*Nullable*	*First*	*Follow*
S	False	{(, **id** , **num** }	$\emptyset$
E	False	{(, **id** , **num** }	$\{), \$\}$
E'	True	$\{+, -\}$	$\{), \$\}$
T	False	{(, **id** , **num** }	$\{), +, -, \$\}$
T'	True	$\{*, /\}$	$\{), +, -, \$\}$
F	False	{(, **id** , **num** }	$\{), *, /, +, -, \$\}$

	$+$	$*$	**id**	(	)	$
S			$S \to E\$$	$S \to E\$$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \epsilon$	$E' \to \epsilon$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to \epsilon$	$T' \to *FT'$			$T' \to \epsilon$	$T' \to \epsilon$
F			$F \to$ **id**	$F \to (E)$		

($-$,/, and **num** are treated similarly)

$LL(1)$ parsing summary so far

Construction of a $LL(1)$ parser from a CFG grammar

- Eliminate ambiguity
- Add an extra start production $S' \rightarrow S\$$ to the grammar
- Calculate *First* for every production and *Follow* for every nonterminal
- Calculate the parsing table
- Check that the grammar is $LL(1)$

Next course:

- Transformations of a grammar to make it $LL(1)$
- Recursive implementation of the predictive parser
- Bottom-up parsing techniques

Transforming a grammar for $LL(1)$ parsing

- Ambiguous grammars are not $LL(1)$ but unambiguous grammars are not necessarily $LL(1)$
- Having a non-$LL(1)$ unambiguous grammar for a language does not mean that this language is not $LL(1)$.
- But there are languages for which there exist unambiguous context-free grammars but no $LL(1)$ grammar.

- We will see two grammar transformations that improve the chance to get a $LL(1)$ grammar:
  - Elimination of left-recursion
  - Left-factorization

Left-recursion

- The following expression grammar is unambiguous but it is not $LL(1)$:

$$
\begin{array}{rcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (Exp)
\end{array}
$$

- Indeed, $First(\alpha)$ is the same for all RHS α of the productions for Exp et $Exp2$
- This is a consequence of *left-recursion*.

Left-recursion

- Recursive productions are productions defined in terms of themselves. Examples: $A \to Ab$ ou $A \to bA$.
- When the recursive nonterminal is at the left (resp. right), the production is said to be left-recursive (resp. right-recursive).
- Left-recursive productions can be rewritten with right-recursive productions
- Example:

$$
\begin{array}{lcl}
N & \to & N\alpha_1 \\
 & \vdots & \\
N & \to & N\alpha_m \\
N & \to & \beta_1 \\
 & \vdots & \\
N & \to & \beta_n
\end{array}
\quad \Leftrightarrow \quad
\begin{array}{lcl}
N & \to & \beta_1 N' \\
 & \vdots & \\
N & \to & \beta_n N' \\
N' & \to & \alpha_1 N' \\
 & \vdots & \\
N' & \to & \alpha_m N' \\
N' & \to & \epsilon
\end{array}
$$

Right-recursive expression grammar

$$
\begin{array}{rcl}
Exp & \to & Exp + Exp2 \\
Exp & \to & Exp - Exp2 \\
Exp & \to & Exp2 \\
Exp2 & \to & Exp2 * Exp3 \\
Exp2 & \to & Exp2 / Exp3 \\
Exp2 & \to & Exp3 \\
Exp3 & \to & \textbf{num} \\
Exp3 & \to & (Exp)
\end{array}
\quad \Leftrightarrow \quad
\begin{array}{rcl}
Exp & \to & Exp2\,Exp' \\
Exp' & \to & +Exp2\,Exp' \\
Exp' & \to & -Exp2\,Exp' \\
Exp' & \to & \epsilon \\
Exp2 & \to & Exp3\,Exp2' \\
Exp2' & \to & *Exp3\,Exp2' \\
Exp2' & \to & /Exp3\,Exp2' \\
Exp2' & \to & \epsilon \\
Exp3 & \to & \textbf{num} \\
Exp3 & \to & (Exp)
\end{array}
$$

Left-factorisation

- The RHS of these two productions have the same *First* set.

$$\begin{aligned} \mathit{Stat} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ else } \mathit{Stat} \\ \mathit{Stat} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \end{aligned}$$

- The problem can be solved by left factorising the grammar:

$$\begin{aligned} \mathit{Stat} &\rightarrow \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat}\ \mathit{ElseStat} \\ \mathit{ElseStat} &\rightarrow \textbf{else } \mathit{Stat} \\ \mathit{ElseStat} &\rightarrow \epsilon \end{aligned}$$

- Note
  - The resulting grammar is ambiguous and the parsing table will contain two rules for $M[\mathit{ElseStat}, \textbf{else}]$
    (because $\textbf{else} \in \mathit{Follow}(\mathit{ElseStat})$ and $\textbf{else} \in \mathit{First}(\textbf{else } \mathit{Stat})$)
  - Ambiguity can be solved in this case by letting
    $M[\mathit{ElseStat}, \textbf{else}] = \{\mathit{ElseStat} \rightarrow \textbf{ else } \mathit{Stat}\}$.

Hidden left-factors and hidden left recursion

- Sometimes, left-factors or left recursion are hidden
- Examples:
  - The following grammar:

$$\begin{aligned} A &\rightarrow da|acB \\ B &\rightarrow abB|daA|Af \end{aligned}$$

    has two overlapping productions: $B \rightarrow daA$ and $B \stackrel{*}{\Rightarrow} daf$.
  - The following grammar:

$$\begin{aligned} S &\rightarrow Tu|wx \\ T &\rightarrow Sq|vvS \end{aligned}$$

    has left recursion on T ($T \stackrel{*}{\Rightarrow} Tuq$)
- Solution: expand the production rules by substitution to make left-recursion or left factors visible and then eliminate them

Summary

Construction of a $LL(1)$ parser from a CFG grammar

- Eliminate ambiguity
- Eliminate left recursion
- left factorization
- Add an extra start production $S' \rightarrow S\$$ to the grammar
- Calculate *First* for every production and *Follow* for every nonterminal
- Calculate the parsing table
- Check that the grammar is $LL(1)$

Recursive implementation

- From the parsing table, it is easy to implement a predictive parser recursively (with one function per nonterminal)

$$
\begin{array}{lcl}
T' & \rightarrow & T\$ \\
T & \rightarrow & R \\
T & \rightarrow & aTc \\
R & \rightarrow & \epsilon \\
R & \rightarrow & bR
\end{array}
$$

	a	b	c	$\$$
T'	$T' \rightarrow T\$$	$T' \rightarrow T\$$		$T' \rightarrow T\$$
T	$T \rightarrow aTc$	$T \rightarrow R$	$T \rightarrow R$	$T \rightarrow R$
R		$R \rightarrow bR$	$R \rightarrow \epsilon$	$R \rightarrow \epsilon$

```
function parseT'() =
  if next = 'a' or next = 'b' or next = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if next = 'b' or next = 'c' or next = '$' then
    parseR()
  else if next = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if next = 'c' or next = '$' then
    (* do nothing *)
  else if next = 'b' then
    match('b') ; parseR()
  else reportError()
```

(Mogensen)

Outline

1. Introduction

2. Context-free grammar

3. Top-down parsing

4. Bottom-up parsing
   - Shift/reduce parsing
   - LR parsers
   - Operator precedence parsing
   - Using ambiguous grammars

5. Conclusion and some practical considerations

Bottom-up parsing

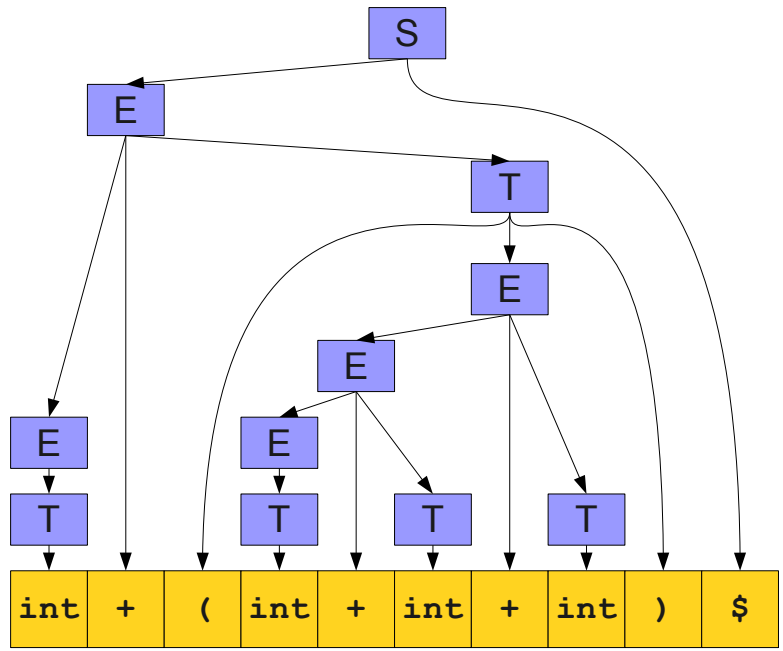
- A bottom-up parser creates the parse tree starting from the leaves towards the root
- It tries to convert the program into the start symbol
- Most common form of bottom-up parsing: shift-reduce parsing

Bottom-up parsing: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E\$ \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of
$int + (int + int + int)$



(Keith Schwarz)

Bottom-up parsing: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E\$ \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of $int + (int + int + int)$:

$int + (int + int + int)\$$
$T + (int + int + int)\$$
$E + (int + int + int)\$$
$E + (T + int + int)\$$
$E + (E + int + int)\$$
$E + (E + T + int)\$$
$E + (E + int)\$$
$E + (E + T)\$$
$E + (E)\$$
$E + T\$$
$E\$$
S

Top-down parsing is often done as a rightmost derivation in reverse (There is only one if the grammar is unambiguous).

Terminology

- A Rightmost (canonical) derivation is a derivation where the rightmost nonterminal is replaced at each step. A rightmost derivation from α to β is noted $\alpha \overset{*}{\Rightarrow}_{rm} \beta$.
- A reduction transforms uwv to uAv if $A \to w$ is a production
- α is a right sentential form if $S \overset{*}{\Rightarrow}_{rm} \alpha$.
- A handle of a right sentential form γ $(= \alpha\beta w)$ is a production $A \to \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ:

$$S \overset{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$$

  - Informally, a handle is a production we can reverse without getting stuck.
  - If the handle is $A \to \beta$, we will also call β the handle.

Handle: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of $int + (int + int + int)$

$$
\begin{array}{l}
int + (int + int + int)\$ \\
T + (int + int + int)\$ \\
E + (int + int + int)\$ \\
E + (T + int + int)\$ \\
E + (E + int + int)\$ \\
E + (E + T + int)\$ \\
E + (E + int)\$ \\
E + (E + T)\$ \\
E + (E)\$ \\
E + T\$ \\
E\$ \\
S
\end{array}
$$

The handle is in red in each right sentential form

Finding the handles

- Bottom-up parsing = finding the handle in the right sentential form obtained at each step
- This handle is unique as soon as the grammar is unambiguous (because in this case, the rightmost derivation is unique)
- Suppose that our current form is uvw and the handle is $A \to v$ (getting uAw after reduction). w can not contain any nonterminals (otherwise we would have reduced a handle somewhere in w)

Shift/reduce parsing

Proposed model for a bottom-up parser:

- Split the input into two parts:
  - Left substring is our work area
  - Right substring is the input we have not yet processed
- All handles are reduced in the left substring
- Right substring consists only of terminals
- At each point, decide whether to:
  - Move a terminal across the split (shift)
  - Reduce a handle (reduce)

Shift/reduce parsing: example

Grammar:

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | \textbf{id}$$

Bottum-up parsing of $id + id * id$

Left substring	Right substring	Action
\$	$id + id * id\$$	Shift
$\$id$	$+id * id\$$	Reduce by $F \rightarrow$ **id**
$\$F$	$+id * id\$$	Reduce by $T \rightarrow F$
$\$T$	$+id * id\$$	Reduce by $E \rightarrow T$
$\$E$	$+id * id\$$	Shift
$\$E+$	$id * id\$$	Shift
$\$E + id$	$*id\$$	Reduce by $F \rightarrow$ **id**
$\$E + F$	$*id\$$	Reduce by $T \rightarrow F$
$\$E + T$	$*id\$$	Shift
$\$E + T*$	$id\$$	Shift
$\$E + T * id$	\$	Reduce by $F \rightarrow$ **id**
$\$E + T * F$	\$	Reduce by $T \rightarrow T * F$
$\$E + T$	\$	Reduce by $E \rightarrow E + T$
$\$E$	\$	Accept

Shift/reduce parsing

- In the previous example, all the handles were to the far right end of the left area (not inside)
- This is convenient because we then never need to shift from the left to the right and thus could process the input from left-to-right in one pass.

- Is it the case for all grammars? Yes !
- Sketch of proof: by induction on the number of reduces
  - After no reduce, the first reduction can be done at the right end of the left area
  - After at least one reduce, the very right of the left area is a nonterminal (by induction hypothesis). This nonterminal must be part or at the left of the next handle, since we are tracing a rightmost derivation backwards.

Shift/reduce parsing

- Consequence: the left area can be represented by a stack (as all activities happen at its far right)

- Four possible actions of a shift-reduce parser:
  1. Shift: push the next terminal onto the stack
  2. Reduce: Replace the handle on the stack by the nonterminal
  3. Accept: parsing is successfully completed
  4. Error: discover a syntax error and call an error recovery routine

Shift/reduce parsing

- There still remain two open questions: At each step:
  - How to choose between shift and reduce?
  - If the decision is to reduce, which rules to choose (i.e., what is the handle)?
- Ideally, we would like this choice to be deterministic given the stack and the next k input symbols (to avoid backtracking), with k typically small (to make parsing efficient)
- Like for top-down parsing, this is not possible for all grammars
- Possible conflicts:
  - shift/reduce conflict: it is not possible to decide between shifting or reducing
  - reduce/reduce conflict: the parser can not decide which of several reductions to make

Shift/reduce parsing

We will see two main categories of shift-reduce parsers:

- LR-parsers
  - They cover a wide range of grammars
  - Different variants from the most specific to the most general: SLR, LALR, LR
- Weak precedence parsers
  - They work only for a small class of grammars
  - They are less efficient than LR-parsers
  - They are simpler to implement

Outline

1. Introduction

2. Context-free grammar

3. Top-down parsing

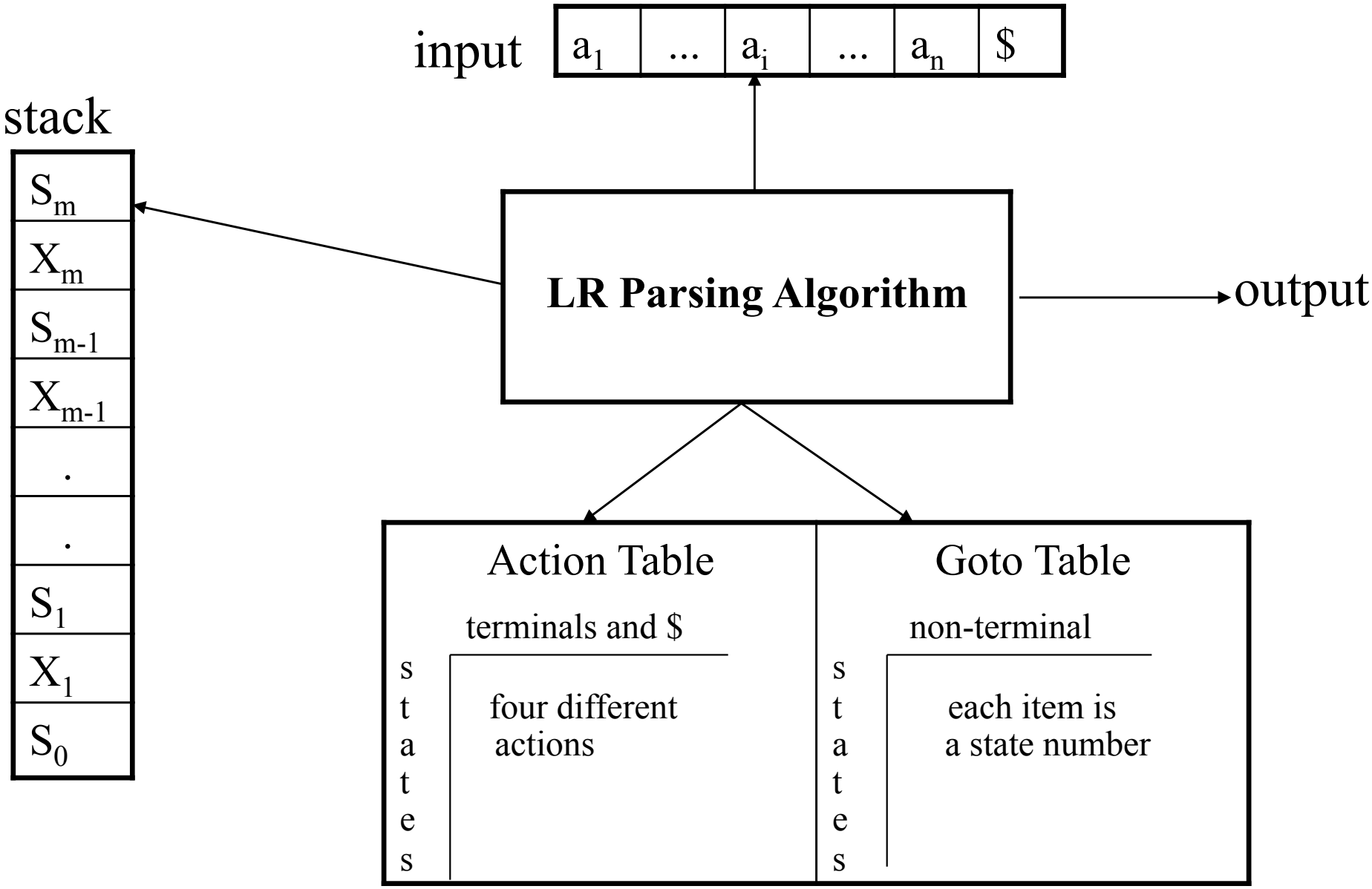
4. Bottom-up parsing
   - Shift/reduce parsing
   - LR parsers
   - Operator precedence parsing
   - Using ambiguous grammars

5. Conclusion and some practical considerations

LR-parsers

- LR(k) parsing: Left-to-right, Rightmost derivation, k symbols lookahead.
- Advantages:
  - The most general non-backtracking shift-reduce parsing, yet as efficient as other less general techniques
  - Can detect syntactic error as soon as possible (on a left-to-right scan of the input)
  - Can recognize virtually all programming language constructs (that can be represented by context-free grammars)
  - Grammars recognized by LR parsers is a proper superset of grammars recognized by predictive parsers (LL(k) $\subset$ LR(k))
- Drawbacks:
  - More complex to implement than predictive (or operator precedence) parsers
- Like table-driven predictive parsing, LR parsing is based on a parsing table.

Structure of a LR parser



Structure of a LR parser

- A configuration of a LR parser is described by the status of its stack and the part of the input not analysed (shifted) yet:

$$(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

  where X_i are (terminal or nonterminal) symbols, a_i are terminal symbols, and s_i are state numbers (of a DFA)
- A configuration corresponds to the right sentential form

$$X_1 \dots X_m a_i \dots a_n$$

- Analysis is based on two tables:
  - an action table that associates an action $ACTION[s, a]$ to each state s and nonterminal a.
  - a goto table that gives the next state $GOTO[s, A]$ from state s after a reduction to a nonterminal A

Actions of a LR-parser

- Let us assume the parser is in configuration

$$(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

(initially, the state is $(s_0, a_1 a_2 \dots a_n \$)$*, where* $a_1 \dots a_n$ *is the input word)*

- $\mathrm{ACTION}[s_m, a_i]$ can take four values:
  1. Shift s: shifts the next input symbol and then the state s on the stack $(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n) \rightarrow (s_0 X_1 s_1 \dots X_m s_m a_i s, a_{i+1} \dots a_n)$
  2. Reduce $A \rightarrow \beta$ (denoted by rn where n is a production number)
     - Pop $2|\beta|$ $(= r)$ items from the stack
     - Push A and s where $s = GOTO[s_{m-r}, A]$
       $(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n) \rightarrow$
       $(s_0 X_1 s_1 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n)$
     - Output the prediction $A \rightarrow \beta$
  3. Accept: parsing is successfully completed
  4. Error: parser detected an error (typically an empty entry in the action table).

LR-parsing algorithm

Create a stack with the start state s_0
$a = \text{GETNEXTTOKEN}()$
while (True)
    $s = \text{POP}()$
    if ($ACTION[s, a] =$ shift t)
        Push a and t onto the stack
        $a = \text{GETNEXTTOKEN}()$
    elseif ($ACTION[s, a] =$ reduce $A \to \beta$)
        Pop $2|\beta|$ elements off the stack
        Let state t now be the state on the top of the stack
        Push A onto the stack
        Push $GOTO[t, A]$ onto the stack
        Output $A \to \beta$
    elseif ($ACTION[s, a] =$ accept)
        break // Parsing is over
    else call error-recovery routine

Example: parsing table for the expression grammar

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

	Action Table						Goto Table		
state	**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: LR parsing with the expression grammar

stack	**input**	**action**	**output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing the parsing tables

- There are several ways of building the parsing tables, among which:
  - LR(0): no lookahead, works for only very few grammars
  - SLR: the simplest one with one symbol lookahead. Works with less grammars than the next ones
  - LR(1): very powerful but generate potentially very large tables
  - LALR(1): tradeoff between the other approaches in terms of power and simplicity
  - LR(k), k$>$ 1: exploit more lookahead symbols

- Main idea of all methods: build a DFA whose states keep track of where we are in the parsing

Parser generators

- LALR(1) is used in most parser generators like Yacc/Bison
- We will nevertheless only see SLR in details:
  - It's simpler.
  - LALR(1) is only minorly more expressive.
  - When a grammar is SLR, then the tables produced by SLR are identical to the ones produced by LALR(1).
  - Understanding of SLR principles is sufficient to understand how to handle a grammar rejected by LALR(1) parser generators (see later).

LR(0) item

- An LR(0) item (or item for short) of a grammar G is a production of G with a dot at some position of the body.
- Example: $A \to XYZ$ yields four items:

$$A \to .XYZ$$
$$A \to X.YZ$$
$$A \to XY.Z$$
$$A \to XYZ.$$

  ($A \to \epsilon$ generates one item $A \to .$)
- An item indicates how much of a production we have seen at a given point in the parsing process.
  - $A \to X.YZ$ means we have just seen on the input a string derivable from X (and we hope to get next YZ).
- Each state of the SLR parser will correspond to a set of LR(0) items
- A particular collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers

Construction of the canonical LR(0) collection

- The grammar G is first augmented into a grammar G' with a new start symbol S' and a production $S' \to S$ where S is the start symbol of G
- We need to define two functions:
  - $\textsc{Closure}(I)$: extends the set of items I when some of them have a dot to the left of a nonterminal
  - $\textsc{Goto}(I, X)$: moves the dot past the symbol X in all items in I
- These two functions will help define a DFA:
  - whose states are (closed) sets of items
  - whose transitions (on terminal and nonterminal symbols) are defined by the $\textsc{Goto}$ function

Closure

Closure(I)

repeat

for any item $A \to \alpha.X\beta$ in I

for any production $X \to \gamma$

$I = I \cup \{X \to .\gamma\}$

until I does not change

return I

Example:

$E' \to E$
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

$$\text{Closure}(\{E' \to .E\}) = \{E' \to .E,$$
$$E \to .E + T$$
$$E \to .T$$
$$T \to .T * F$$
$$T \to .F$$
$$F \to .(E)$$
$$F \to .\ \textbf{id}\ \}$$

Goto

$\textsc{Goto}(I, X)$

Set J to the empty set
for any item $A \to \alpha.X\beta$ in I
$J = J \bigcup \{A \to \alpha X.\beta\}$
return $\textsc{closure}(J)$

Example:

$E' \to E$
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

$I_0 = \{E' \to .E,$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .$ **id** $\}$

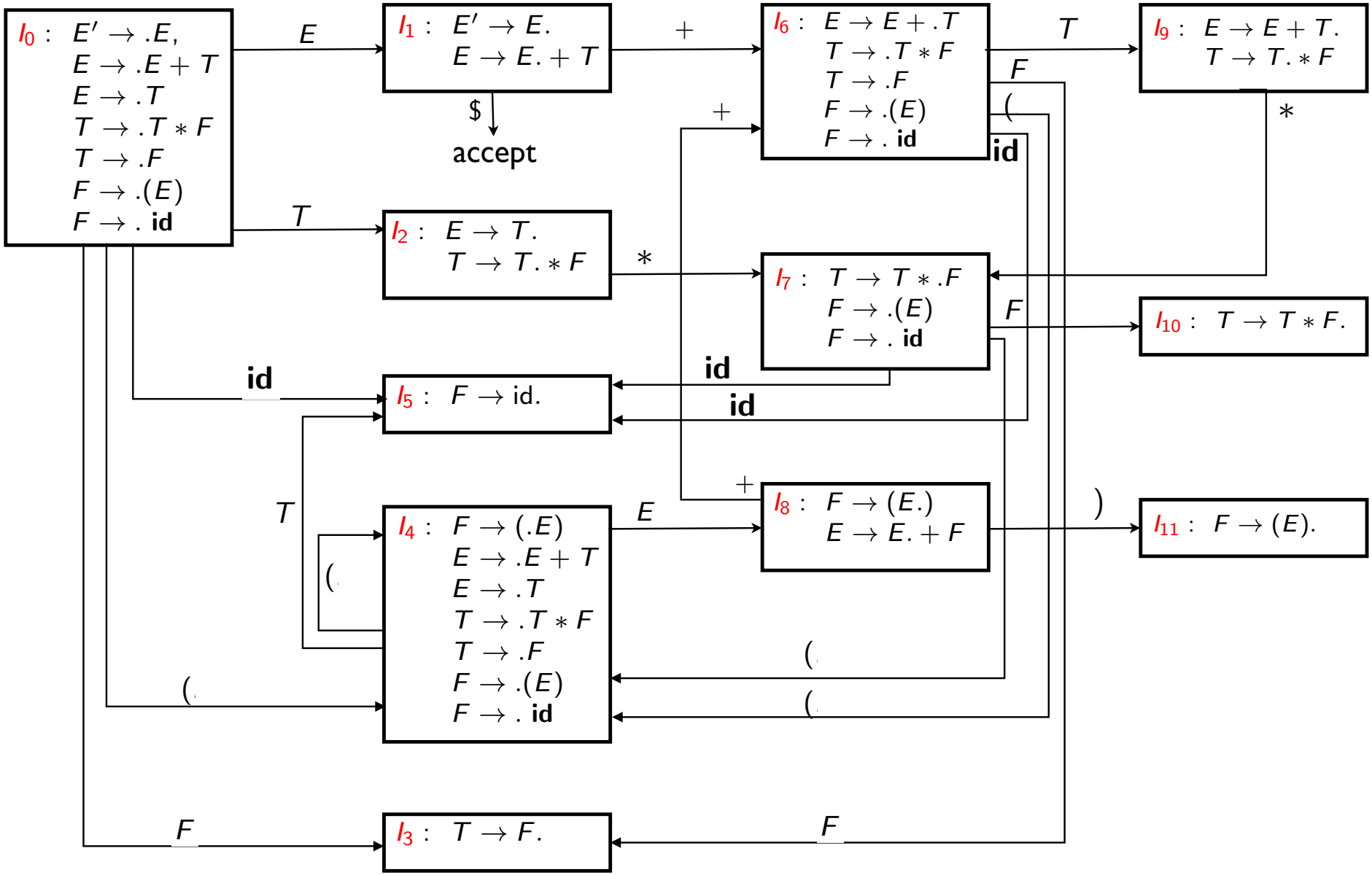
$\textsc{goto}(I_0, E) = \{E' \to E., E \to E. + T\}$
$\textsc{goto}(I_0, T) = \{E \to T., T \to T. * F\}$
$\textsc{goto}(I_0, F) = \{T \to F.\}$
$\textsc{goto}(I_0,'(') = \textsc{Closure}(\{F \to (.E)\})$
$= \{F \to (.E)\} \cup (I_0 \setminus \{E' \to E\})$
$\textsc{goto}(I_0, \mathsf{id}) = \{F \to \mathsf{id}.\}$

Construction of the canonical collection

$C = \{\textsc{closure}(\{S' \to .S\})\}$
repeat
  for each item set I in C
    for each item $A \to \alpha.X\beta$ in I
      $C = C \cup \{\textsc{Goto}(I, X)\}$
until C did not change in this iteration
return C

- Collect all sets of items reachable from the initial state by one or several applications of GOTO.
- Item sets in C are the states of a DFA, GOTO is its transition function

Example


I_0 : $E' \to .E$, $E \to .E + T$, $E \to .T$, $T \to .T * F$, $T \to .F$, $F \to .(E)$, $F \to .$ id
I_1 : $E' \to E.$, $E \to E. + T$
I_2 : $E \to T.$, $T \to T. * F$
I_3 : $T \to F.$
I_4 : $F \to (.E)$, $E \to .E + T$, $E \to .T$, $T \to .T * F$, $T \to .F$, $F \to .(E)$, $F \to .$ id
I_5 : $F \to$ id.
I_6 : $E \to E + .T$, $T \to .T * F$, $T \to .F$, $F \to .(E)$, $F \to .$ id
I_7 : $T \to T * .F$, $F \to .(E)$, $F \to .$ id
I_8 : $F \to (E.)$, $E \to E. + F$
I_9 : $E \to E + T.$, $T \to T. * F$
I_{10} : $T \to T * F.$
I_{11} : $F \to (E).$
accept
E
T
F
id
+
*
(
)
$

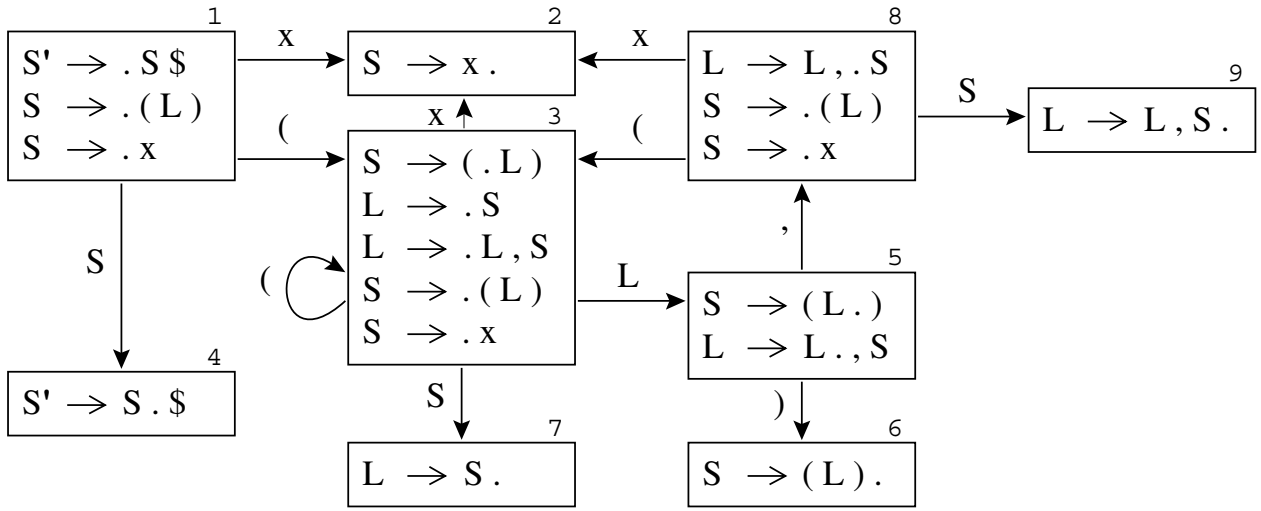
Constructing the LR(0) parsing table

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of $LR(0)$ items for G' (the augmented grammar)
2. State i of the parser is derived from I_i. Actions for state i are as follows:
   - 2.1 If $A \to \alpha.a\beta$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] = \text{Shift } j$
   - 2.2 If $A \to \alpha.$ is in I_i, then set $ACTION[i, a] = \text{Reduce } A \to \alpha$ for all terminals a.
   - 2.3 If $S' \to S.$ is in I_i, then set $ACTION[i, \$] = \text{Accept}$
3. If $\text{GOTO}(I_i, X) = I_j$, then $GOTO[i, X] = j$.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state s_0 is the set of items containing $S' \to .S$

$\Rightarrow$ *LR(0) because the chosen action (shift or reduce) only depends on the current state (but the choice of the next state still depends on the token)*

Example of a LR(0) grammar

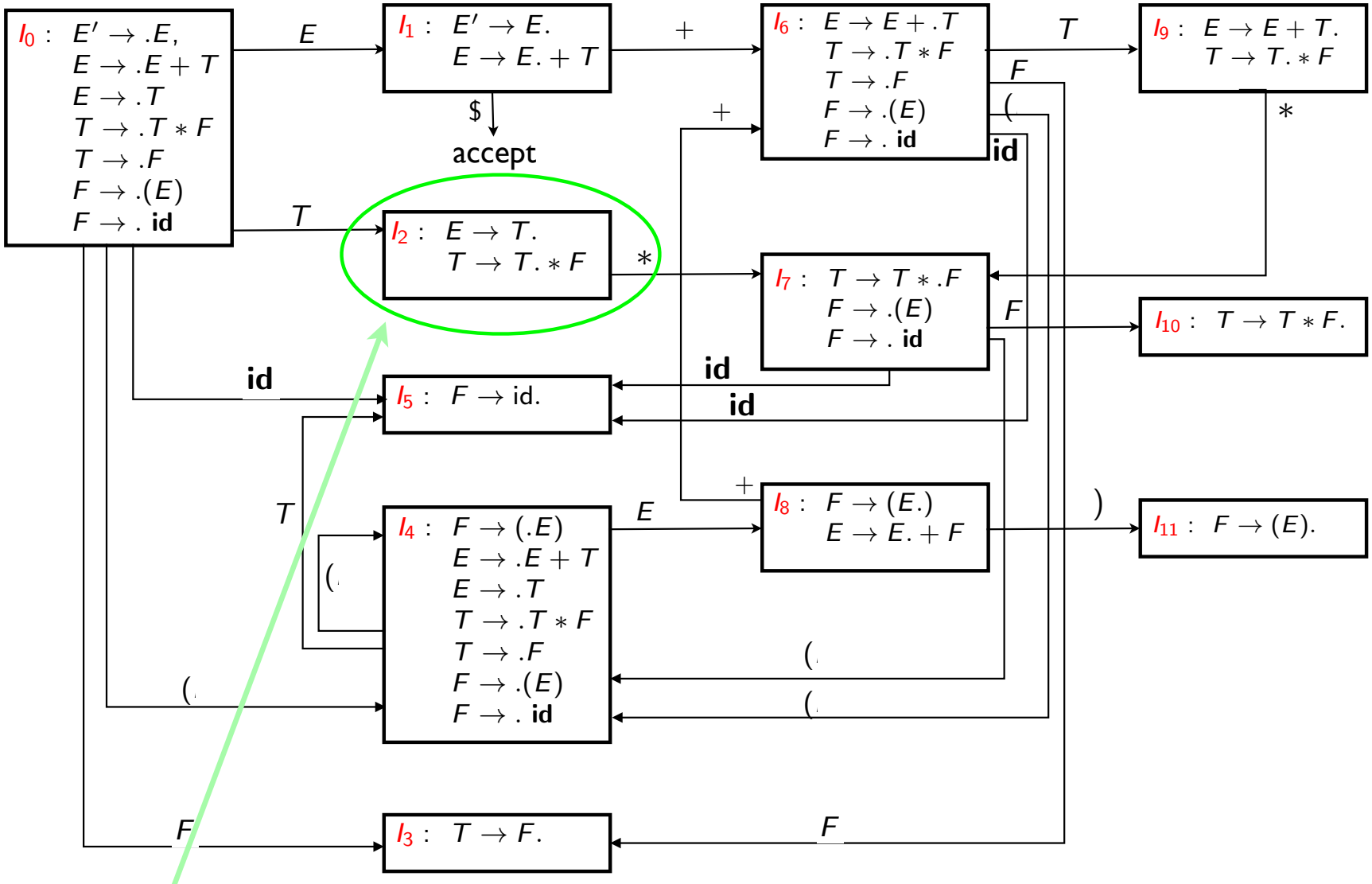
0 $S' \rightarrow S\$$
1 $S \rightarrow (L)$
2 $S \rightarrow x$
3 $L \rightarrow S$
4 $L \rightarrow L , S$



	(	)	x	,	$	*S*	*L*
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

(Appel)

Example of a non LR(0) grammar



Conflict: in state 2, we don't know whether to shift or reduce.

Constructing the SLR parsing tables

1. Construct $c = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of $LR(0)$ items for G' (the augmented grammar)
2. State i of the parser is derived from I_i. Actions for state i are as follows:
   - 2.1 If $A \to \alpha.a\beta$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] =$ Shift j
   - 2.2 If $A \to \alpha.$ is in I_i, then $ACTION[i, a] =$ Reduce $A \to \alpha$ for all terminals a in $Follow(A)$ where $A \neq S'$
   - 2.3 If $S' \to S.$ is in I_i, then set $ACTION[i, \$] =$ Accept
3. If $\text{GOTO}(I_i, A) = I_j$ for a nonterminal A, then $GOTO[i, A] = j$
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state s_0 is the set of items containing $S' \to .S$

$\Rightarrow$ *the simplest form of one symbol lookahead, SLR (Simple LR)*

Example

	First	*Follow*
E	**id** (	$ +)
T	**id** (	$ + *)
F	**id** (	$ + *)

	Action Table							Goto Table		
state	**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				

SLR(1) grammars

- A grammar for which there is no (shift/reduce or reduce/reduce) conflict during the construction of the SLR table is called SLR(1) (or SLR in short).
- All SLR grammars are unambiguous but many unambiguous grammars are not SLR
- There are more SLR grammars than LL(1) grammars but there are LL(1) grammars that are not SLR.

Conflict example for SLR parsing

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

$$
\begin{array}{ll}
I_0: & S' \rightarrow \cdot S \\
& S \rightarrow \cdot L = R \\
& S \rightarrow \cdot R \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id} \\
& R \rightarrow \cdot L \\
\\
I_1: & S' \rightarrow S \cdot \\
\\
I_2: & S \rightarrow L \cdot = R \\
& R \rightarrow L \cdot \\
\\
I_3: & S \rightarrow R \cdot \\
\\
I_4: & L \rightarrow * \cdot R \\
& R \rightarrow \cdot L \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id}
\end{array}
$$

$$
\begin{array}{ll}
I_5: & L \rightarrow \mathbf{id} \cdot \\
\\
I_6: & S \rightarrow L = \cdot R \\
& R \rightarrow \cdot L \\
& L \rightarrow \cdot * R \\
& L \rightarrow \cdot \mathbf{id} \\
\\
I_7: & L \rightarrow *R \cdot \\
\\
I_8: & R \rightarrow L \cdot \\
\\
I_9: & S \rightarrow L = R \cdot
\end{array}
$$

(Dragonbook)

Follow(R) contains '='. In I_2, when seeing '=' on the input, we don't know whether to shift or to reduce with $R \rightarrow L$.

Summary of SLR parsing

Construction of a SLR parser from a CFG grammar

- Eliminate ambiguity *(or not, see later)*
- Add the production $S' \to S$, where S is the start symbol of the grammar
- Compute the LR(0) canonical collection of LR(0) item sets and the GOTO function (transition function)
- Add a shift action in the action table for transitions on terminals and goto actions in the goto table for transitions on nonterminals
- Compute *Follow* for each nonterminals (which implies first adding $S'' \to S'\$$ to the grammar and computing *First* and *Nullable*)
- Add the reduce actions in the action table according to *Follow*
- Check that the grammar is SLR *(and if not, try to resolve conflicts, see later)*

Outline

1. Introduction

2. Context-free grammar

3. Top-down parsing

4. Bottom-up parsing
   - Shift/reduce parsing
   - LR parsers
   - Operator precedence parsing
   - Using ambiguous grammars

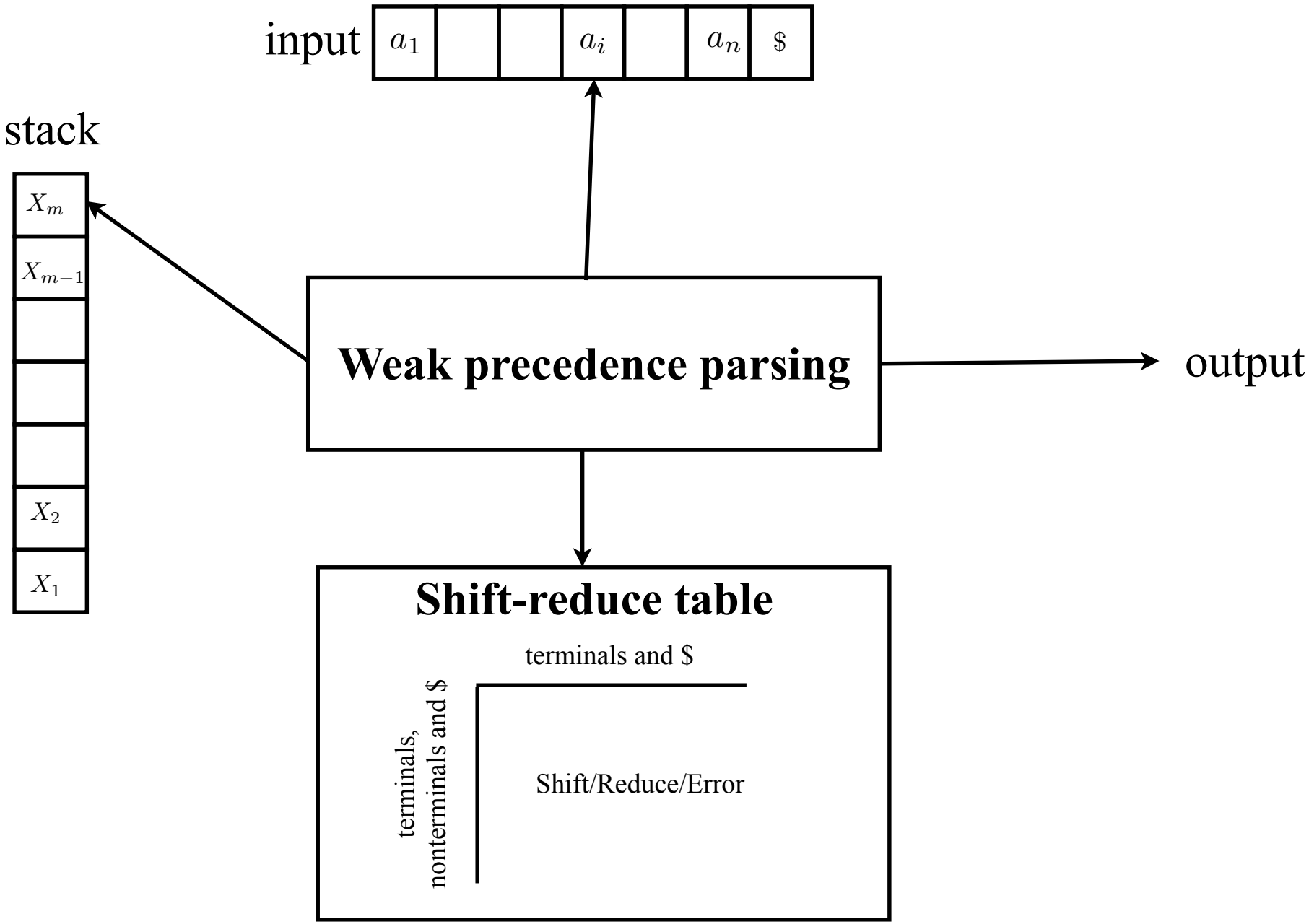
5. Conclusion and some practical considerations

Operator precedence parsing

- Bottom-up parsing methods that follow the idea of shift-reduce parsers
- Several flavors: operator, simple, and weak precedence.
- In this course, only weak precedence

- Main differences compared to LR parsers:
  - There is no explicit state associated to the parser (and thus no state pushed on the stack)
  - The decision of whether to shift or reduce is taken based solely on the symbol on the top of the stack and the next input symbol (and stored in a shift-reduce table)
  - In case of reduction, the handle is the longest sequence at the top of stack matching the RHS of a rule

Structure of the weak precedence parser



Weak precedence parsing algorithm

Create a stack with the special symbol \$
a = GETNEXTTOKEN()
while (True)
    if (Stack== \$$S$ and a == \$)
        break // Parsing is over
    X_m = TOP(Stack)
    if ($SRT[X_m, a]$ = shift)
        Push a onto the stack
        a = GETNEXTTOKEN()
    elseif ($SRT[X_m, a]$ = reduce)
        Search for the longest RHS that matches the top of the stack
        if no match found
            call error-recovery routine
        Let denote this rule by $Y \rightarrow X_{m-r+1} \ldots X_m$
        Pop r elements off the stack
        Push Y onto the stack
        Output $Y \rightarrow X_{m-r+1} \ldots X_m$
    else call error-recovery routine

Example for the expression grammar

Example:

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Shift/reduce table

	$*$	$+$	(	)	**id**	$
E		S		S		R
T	S	R		R		R
F	R	R		R		R
$*$			S		S	
$+$			S		S	
(			S		S	
)	R	R		R		R
id	R	R		R		R
$			S		S	

Example of parsing

Stack	Input	Action
$\$$	$id + id * id\$$	Shift
$\$id$	$+id * id\$$	Reduce by $F \rightarrow$ **id**
$\$F$	$+id * id\$$	Reduce by $T \rightarrow F$
$\$T$	$+id * id\$$	Reduce by $E \rightarrow T$
$\$E$	$+id * id\$$	Shift
$\$E+$	$id * id\$$	Shift
$\$E + id$	$*id\$$	Reduce by $F \rightarrow$ **id**
$\$E + F$	$*id\$$	Reduce by $T \rightarrow F$
$\$E + T$	$*id\$$	Shift
$\$E + T*$	$id\$$	Shift
$\$E + T * id$	$\$$	Reduce by $F \rightarrow$ **id**
$\$E + T * F$	$\$$	Reduce by $T \rightarrow T * F$
$\$E + T$	$\$$	Reduce by $E \rightarrow E + T$
$\$E$	$\$$	Accept

Precedence relation: principle

- We define the (weak precedence) relations $\lessdot$ and $\gtrdot$ between symbols of the grammar (terminals or nonterminals)
  - $X \lessdot Y$ if XY appears in the RHS of a rule or if X precedes a reducible word whose leftmost symbol is Y
  - $X \gtrdot Y$ if X is the rightmost symbol of a reducible word and Y the symbol immediately following that word
- Shift when $X_m \lessdot a$, reduce when $X_m \gtrdot a$
- Reducing changes the precedence relation only at the top of the stack (there is thus no need to shift backward)

Precedence relation: formal definition

- Let $G = (V, \Sigma, R, S)$ be a context-free grammar and \$ a new symbol acting as left and right end-marker for the input word. Define $V' = V \cup \{\$\}$
- The weak precedence relations $\lessdot$ and $\gtrdot$ are defined respectively on $V' \times V$ and $V \times V'$ as follows:
  1. $X \lessdot Y$ if $A \to \alpha X B \beta$ is in R, and $B \stackrel{+}{\Rightarrow} Y\gamma$,
  2. $X \lessdot Y$ if $A \to \alpha X Y \beta$ is in R
  3. $\$ \lessdot X$ if $S \stackrel{+}{\Rightarrow} X\alpha$
  4. $X \gtrdot a$ if $A \to \alpha B \beta$ is in R, and $B \stackrel{+}{\Rightarrow} \gamma X$ and $\beta \stackrel{*}{\Rightarrow} a\gamma$
  5. $X \gtrdot \$$ if $S \stackrel{+}{\Rightarrow} \alpha X$

  for some α, β, γ, and B

Construction of the SR table: shift

Shift relation, $\lessdot$:

Initialize $\mathcal{S}$ to the empty set.
1 add $\$ \lessdot S$ to $\mathcal{S}$
2 **for** each production $X \to L_1 L_2 \ldots L_k$
  for $i = 1$ **to** $k - 1$
    add $L_i \lessdot L_{i+1}$ to $\mathcal{S}$
3 **repeat**
  for each* pair $X \lessdot Y$ in $\mathcal{S}$
    for each production $Y \to L_1 L_2 \ldots L_k$
      Add $X \lessdot L_1$ to $\mathcal{S}$
until $\mathcal{S}$ did not change in this iteration.

* We only need to consider the pairs $X \lessdot Y$ with Y a nonterminal that were added in $\mathcal{S}$ at the previous iteration

Example of the expression grammar: shift

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Step 1	$S \lessdot \$$
Step 2	$E \lessdot +$
	$+ \lessdot T$
	$T \lessdot *$
	$* \lessdot F$
	$(\lessdot E$
	$E \lessdot)$
Step 3.1	$+ \lessdot F$
	$* \lessdot$ **id**
	$* \lessdot ($
	$(\lessdot T$
Step 3.2	$+ \lessdot$ **id**
	$+ \lessdot ($
	$(\lessdot F$
Step 3.3	$(\lessdot ($
	$(\lessdot$ **id**

Construction of the SR table: reduce

Reduce relation, $\gtrdot$:

Initialize $\mathcal{R}$ to the empty set.
1 add $S \gtrdot \$$ to $\mathcal{R}$
2 **for** each production $X \to L_1 L_2 \ldots L_k$
  for each pair $X \lessdot Y$ in $\mathcal{S}$
    add $L_k \gtrdot Y$ in $\mathcal{R}$
3 **repeat**
  for each* pair $X \gtrdot Y$ in $\mathcal{R}$
    for each production $X \to L_1 L_2 \ldots L_k$
      Add $L_k \gtrdot Y$ to $\mathcal{R}$
until $\mathcal{R}$ did not change in this iteration.

* We only need to consider the pairs $X \gtrdot Y$ with X a nonterminal that were added in $\mathcal{R}$ at the previous iteration.

Example of the expression grammar: reduce

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Step 1	$E \gtrdot \$$
Step 2	$T \gtrdot +$
	$F \gtrdot *$
	$T \gtrdot)$
Step 3.1	$T \gtrdot \$$
	$F \gtrdot +$
	$) \gtrdot *$
	id $\gtrdot *$
	$F \gtrdot)$
Step 3.2	$F \gtrdot \$$
	$) \gtrdot +$
	id $\gtrdot +$
	$) \gtrdot)$
	id $\gtrdot)$
Step 3.3	**id** $\gtrdot \$$
	$) \gtrdot \$$

Weak precedence grammars

- Weak precedence grammars are those that can be analysed by a weak precedence parser.
- A grammar $G = (V, \Sigma, R, S)$ is called a weak precedence grammar if it satisfies the following conditions:
  1. There exist no pair of productions with the same right hand side
  2. There are no empty right hand sides ($A \rightarrow \epsilon$)
  3. There is at most one weak precedence relation between any two symbols
  4. Whenever there are two syntactic rules of the form $A \rightarrow \alpha X \beta$ and $B \rightarrow \beta$, we don't have $X \lessdot B$
- Conditions 1 and 2 are easy to check
- Conditions 3 and 4 can be checked by constructing the SR table.

Example of the expression grammar

$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to$ **id**

Shift/reduce table

	$*$	$+$	(	)	**id**	$
E		S		S		R
T	S	R		R		R
F	R	R		R		R
$*$			S		S	
$+$			S		S	
(			S		S	
)	R	R		R		R
id	R	R		R		R
$			S		S	

- Conditions 1-3 are satisfied (there is no conflict in the SR table)
- Condition 4:
  - $E \to E + T$ and $E \to T$ but we don't have $+ \lessdot E$ (see slide 202)
  - $T \to T * F$ and $T \to F$ but we don't have $* \lessdot T$ (see slide 202)

Removing ϵ rules

- Removing rules of the form $A \to \epsilon$ is not difficult
- For each rule with A in the RHS, add a set of new rules consisting of the different combinations of A replaced or not with ϵ.
- Example:

$$\begin{aligned} S &\to AbA|B \\ B &\to b|c \\ A &\to \epsilon \end{aligned}$$

is transformed into

$$\begin{aligned} S &\to AbA|Ab|bA|b|B \\ B &\to b|c \end{aligned}$$

Summary of weak precedence parsing

Construction of a weak precedence parser

- Eliminate ambiguity *(or not, see later)*
- Eliminate productions with ϵ and ensure that there are no two productions with identical RHS
- Construct the shift/reduce table
- Check that there is no conflict during the construction
- Check condition 4 of slide 205

Outline

1. Introduction

2. Context-free grammar

3. Top-down parsing

4. Bottom-up parsing
   - Shift/reduce parsing
   - LR parsers
   - Operator precedence parsing
   - Using ambiguous grammars

5. Conclusion and some practical considerations

Using ambiguous grammars with bottom-up parsers

- All grammars used in the construction of Shift/Reduce parsing tables must be un-ambiguous
- We can still create a parsing table for an ambiguous grammar but there will be conflicts
- We can often resolve these conflicts in favor of one of the choices to disambiguate the grammar
- Why use an ambiguous grammar?
  - Because the ambiguous grammar is much more natural and the corresponding unambiguous one can be very complex
  - Using an ambiguous grammar may eliminate unnecessary reductions
- Example:

$$E \to E + E | E * E | (E) | \mathbf{id} \quad \Rightarrow \quad \begin{array}{l} E \to E + T | T \\ T \to T * F | F \\ F \to (E) | \mathbf{id} \end{array}$$

Set of LR(0) items of the ambiguous expression grammar

$E \to E + E | E * E | (E) | \mathbf{id}$

$Follow(E) = \{\$, +, *,)\}$
$\Rightarrow$ states 7 and 8 have shift/reduce conflicts for $+$ and $*$.

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id} \cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

(Dragonbook)

Disambiguation

Example:

- Parsing of $\mathbf{id} + \mathbf{id} * \mathbf{id}$ will give the configuration

$$(0E1 + 4E7, *\mathbf{id}\$)$$

We can choose:

- $ACTION[7, *]$ =shift 5$\Rightarrow$ precedence to $*$
- $ACTION[7, *]$ =reduce $E \to E + E \Rightarrow$ precedence to $+$

- Parsing of $\mathbf{id} + \mathbf{id} + \mathbf{id}$ will give the configuration

$$(0E1 + 4E7, +\mathbf{id}\$)$$

We can choose:

- $ACTION[7, +]$ =shift 4$\Rightarrow$ $+$ is right-associative
- $ACTION[7, +]$ =reduce $E \to E + E \Rightarrow$ $+$ is left-associative

(same analysis for I_8)

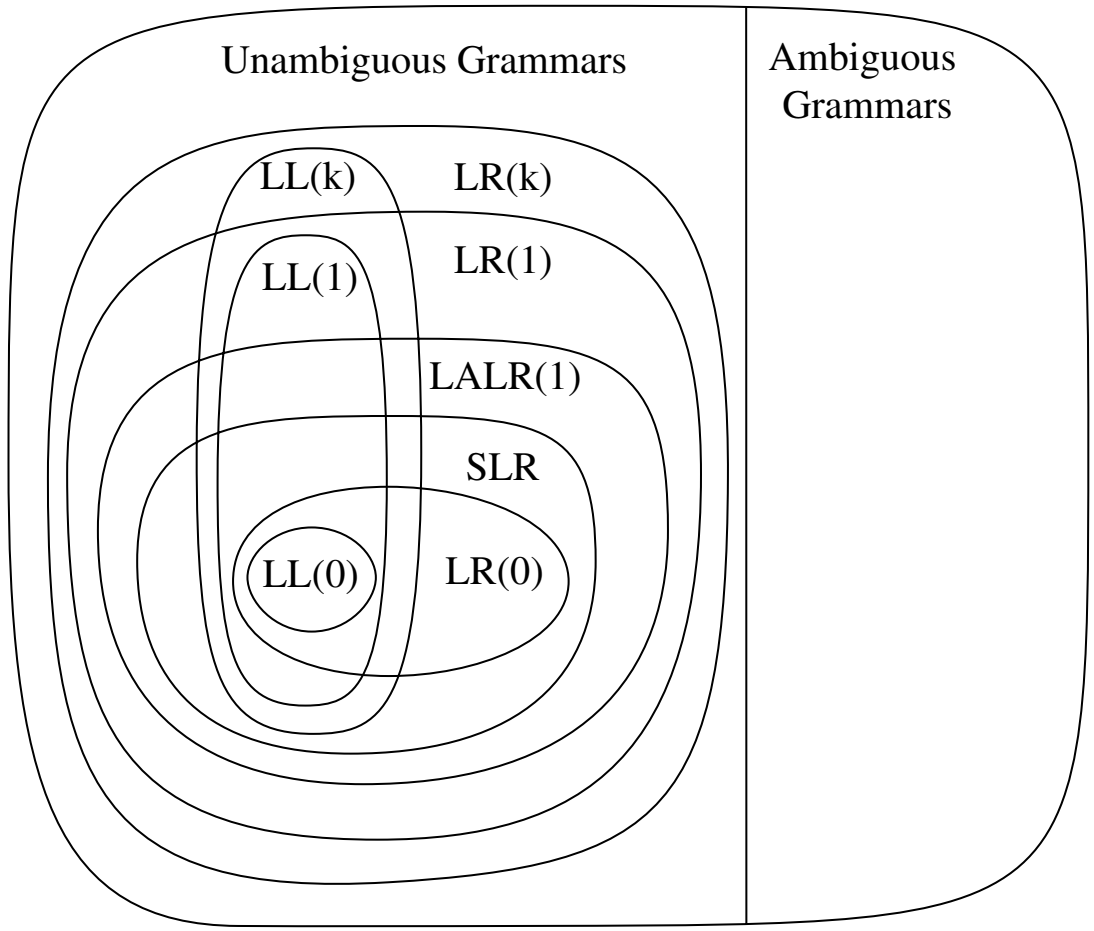
outline

1. Introduction
2. Context-free grammar
3. Top-down parsing
4. Bottom-up parsing
    - Shift/reduce parsing
    - LR parsers
    - Operator precedence parsing
    - Using ambiguous grammars
5. Conclusion and some practical considerations

Top-down versus bottom-up parsing

- Top-down
  - Easier to implement (recursively), enough for most standard programming languages
  - Need to modify the grammar sometimes strongly, less general than bottom-up parsers
  - Used in most hand-written compilers and some parser generators (JavaCC, ANTLR)
- Bottom-up:
  - More general, less strict rules on the grammar, SLR(1) powerful enough for most standard programming languages
  - More difficult to implement, less easy to maintain (add new rules, etc.)
  - Used in most parser generators (Yacc, Bison)

Hierarchy of grammar classes



(Appel)

Error detection and recovery

- In table-driven parsers, there is an error as soon as the table contains no entry (or an error entry) for the current stack (state) and input symbols
- The least one can do: report a syntax error and give information about the position in the input file and the tokens that were expected at that position
- In practice, it is however desirable to continue parsing to report more errors
- There are several ways to recover from an error:
  - Panic mode
  - Phrase-level recovery
  - Introduce specific productions for errors
  - Global error repair

Panic-mode recovery

- In case of syntax error within a "phrase", skip until the next synchronizing token is found (e.g., semicolon, right parenthesis) and then resume parsing
- In LR parsing:
  - Scan down the stack until a state s with a goto on a particular nonterminal A is found
  - Discard zero or more input symbols until a symbol a is found that can follow A
  - Stack the state $GOTO(s, A)$ and resume normal parsing

Phrase-level recovery

- Examine each error entry in the parsing table and decide on an appropriate recovery procedure based on the most likely programmer error.
- Examples in LR parsing: $E \rightarrow E + E | E * E | (E) | id$
  - $id + *id$:
    $*$ is unexpected after a $+$: report a "missing operand" error, push an arbitrary number on the stack and go to the appropriate next state
  - $id + id) + id$:
    Report an "unbalanced right parenthesis" error and remove the right parenthesis from the input

Other error recovery approaches

Introduce specific productions for detecting errors:

- Add rules in the grammar to detect common errors
- Examples for a *C* compiler:
  $I \rightarrow$ **if** E I (parenthesis are missing around the expression)
  $I \rightarrow$ **if** (E) **then** I (**then** is not needed in C)

Global error repair:

- Try to find *globally* the smallest set of insertions and deletions that would turn the program into a syntactically correct string
- Very costly and not always effective

Building the syntax tree

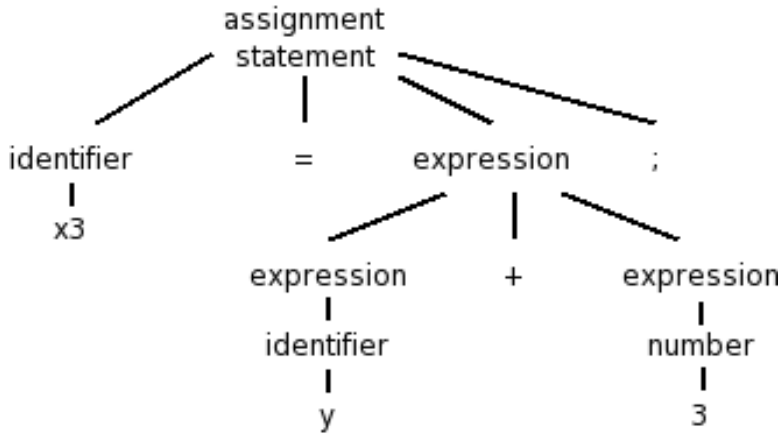
- Parsing algorithms presented so far only check that the program is syntactically correct
- In practice, the parser also needs to build the parse tree (also called concrete syntax tree)
- Its construction is easily embedded into the parsing algorithm

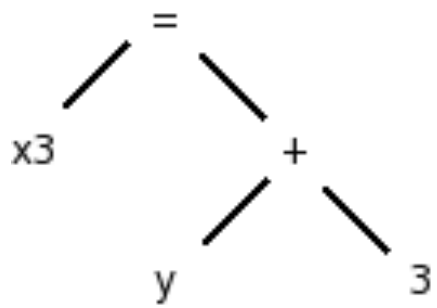
- Top-down parsing:
  - Recursive descent: let each parsing function return the sub-trees for the parts of the input they parse
  - Table-driven: each nonterminal on the stack points to its node in the partially built syntax tree. When the nonterminal is replaced by one of its RHS, nodes for the symbols on the RHS are added as children to the nonterminal node

Building the syntax tree

- Bottom-up parsing:
  - Each stack element points to a subtree of the syntax tree
  - When performing a reduce, a new syntax tree is built with the nonterminal at the root and the popped-off stack elements as children

- Note:
  - In practice, the concrete syntax tree is not built but rather a simplified (abstract) syntax tree
  - Depending on the complexity of the compiler, the syntax tree might even not be constructed





For your project

- The choice of a parsing technique is left open for the project
- You can either use a parser generator or implement the parser by yourself
- Motivate your choice in your report and explain any transformation you had to apply to your grammar to make it fit the constraints of the parser

- Parser generators:
  - Yacc: Unix parser generator, LALR(1) (companion of Lex)
  - Bison: free implementation of Yacc, LALR(1) (companion of Flex)
  - ANTLR: LL(*), implemented in Java but output code in several languages
  - ...
- `http://en.wikipedia.org/wiki/Comparison_of_parser_generators`

An example with Flex/Bison

Example: Parsing of the following expression grammar:

$$
\begin{aligned}
\mathit{Input} &\rightarrow \mathit{Input}\ \mathit{Line} \\
\mathit{Input} &\rightarrow \epsilon \\
\mathit{Line} &\rightarrow \mathit{Exp}\ \mathit{EOL} \\
\mathit{Line} &\rightarrow \mathit{EOL} \\
\mathit{Exp} &\rightarrow \textbf{num} \\
\mathit{Exp} &\rightarrow \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp} - \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp} * \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp}/\mathit{Exp} \\
\mathit{Exp} &\rightarrow (\mathit{Exp})
\end{aligned}
$$

`https://github.com/prashants/calc`

Flex file: calc.lex

```
%{
#define YYSTYPE double /* Define the main semantic type */
#include "calc.tab.h"  /* Define the token constants    */
#include <stdlib.h>
%}
%option yylineno       /* Ask flex to put line number in yylineno */
white [ \t]+
digit [0-9]
integer {digit}+
exponent [eE][+-]?{integer}
real {integer}("."{integer})?{exponent}?
%%
{white} {}
{real} { yylval=atof(yytext); return NUMBER; }
"+" { return PLUS; }
"-" { return MINUS; }
"*" { return TIMES; }
"/" { return DIVIDE; }
"(" { return LEFT; }
")" { return RIGHT; }
"\n" { return END; }
. { yyerror("Invalid token"); }
```

Bison file: calc.y

- Declaration:

```
%{
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define YYSTYPE double  /* Define the main semantic type */
extern char *yytext;    /* Global variables of Flex */
extern int yylineno;
extern FILE *yyin;
%}
```

- Definition of the tokens and start symbol

```
%token NUMBER
%token PLUS MINUS TIMES DIVIDE
%token LEFT RIGHT
%token END

%start Input
```

Bison file: calc.y

- Operator associativity and precedence:

```
%left PLUS MINUS
%left TIMES DIVIDE
%left NEG
```

- Production rules and associated actions:

```
%%

Input:                    /* epsilon */
        | Input Line
;

Line:
        END
        | Expression END { printf("Result: %f\n", $1); }
;
```

Bison file: calc.y

- Production rules and actions (continued):

```
Expression:
        NUMBER { $$ = $1; }
        | Expression PLUS Expression { $$ = $1 + $3; }
        | Expression MINUS Expression { $$ = $1 - $3; }
        | Expression TIMES Expression { $$ = $1 * $3; }
        | Expression DIVIDE Expression { $$ = $1 / $3; }
        | MINUS Expression %prec NEG { $$ = -$2; }
        | LEFT Expression RIGHT { $$ = $2; }
;
```

- Error handling:

```
%%

int yyerror(char *s)
{
        printf("%s on line %d - %s\n", s, yylineno, yytext);
}
```

Bison file: calc.y

- Main functions:

```
int main(int argc, char **argv)
{
   /* if any input file has been specified read from that */
   if (argc >= 2) {
      yyin = fopen(argv[1], "r");
      if (!yyin) {
         fprintf(stderr, "Failed to open input file\n");
      }
      return EXIT_FAILURE;
   }

   if (yyparse()) {
      fprintf(stdout, "Successful parsing\n");
   }

   fclose(yyin);
   fprintf(stdout, "End of processing\n");
   return EXIT_SUCCESS;
}
```

Bison file: makefile

- How to compile:

```
bison -v -d calc.y
flex -o calc.lex.c calc.lex
gcc -o calc calc.lex.c calc.tab.c -lfl -lm
```

- Example:

```
>./calc
1+2*3-4
Result: 3.000000
1+3*-4
Result: -11.000000
*2
syntax error on line 3 - *
Successful parsing
End of processing
```

The state machine

Excerpt of calc.output (with *Expression* abbreviated in *Exp*):

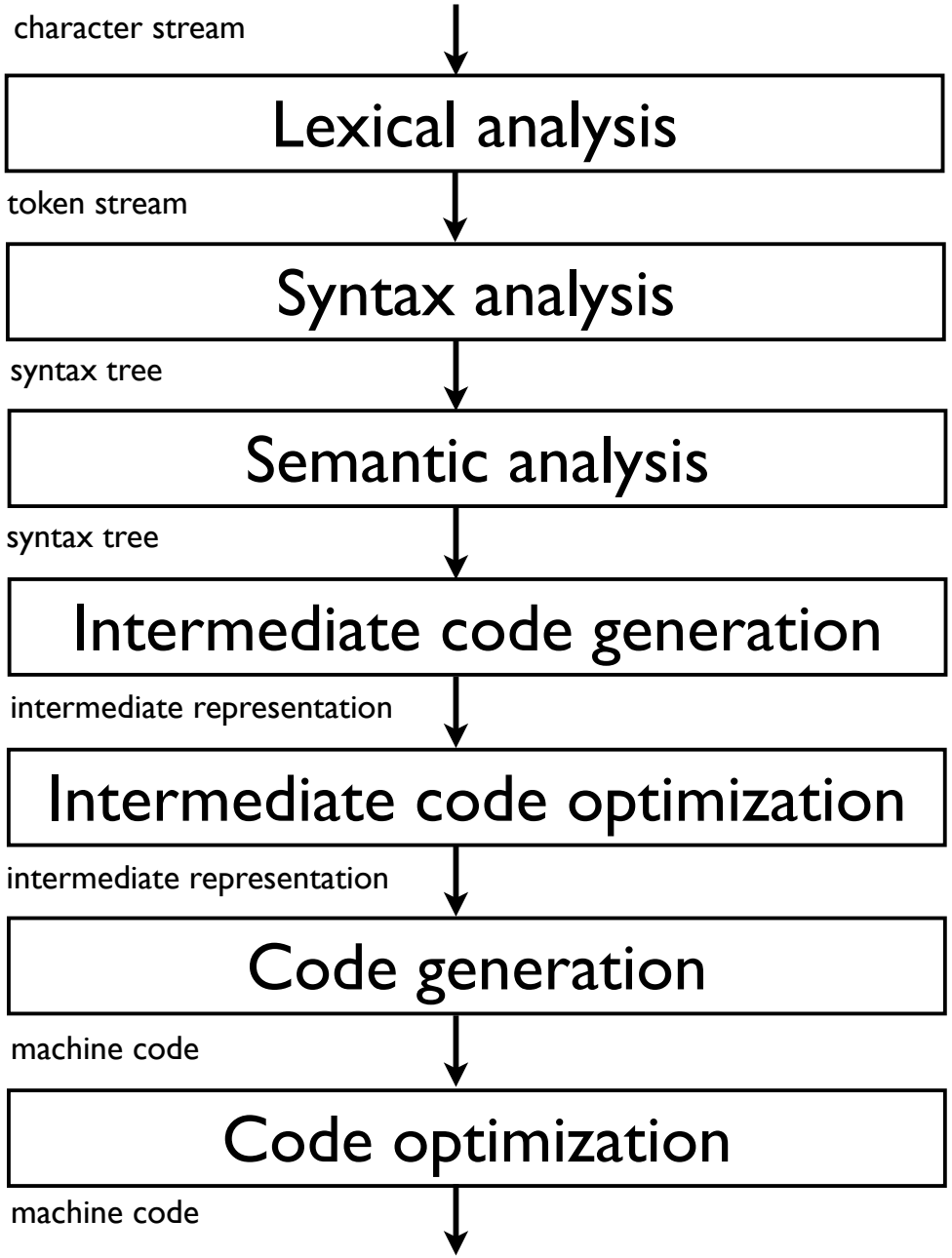
```
state 9
    6 Exp: Exp . PLUS Exp
    7    | Exp . MINUS Exp
    8    | Exp . TIMES Exp
    9    | Exp . DIVIDE Exp
   10    | MINUS Exp .

    $default  reduce using rule 10 (Exp)

state 10
    6 Exp: Exp . PLUS Exp
    7    | Exp . MINUS Exp
    8    | Exp . TIMES Exp
    9    | Exp . DIVIDE Exp
   11    | LEFT Exp . RIGHT

    PLUS    shift, and go to state 11
    MINUS   shift, and go to state 12
    TIMES   shift, and go to state 13
    DIVIDE  shift, and go to state 14
    RIGHT   shift, and go to state 16
```

```
state 11
    6 Exp: Exp PLUS . Exp

    NUMBER  shift, and go to state 3
    MINUS   shift, and go to state 4
    LEFT    shift, and go to state 5

    Exp  go to state 17
```

Part 4

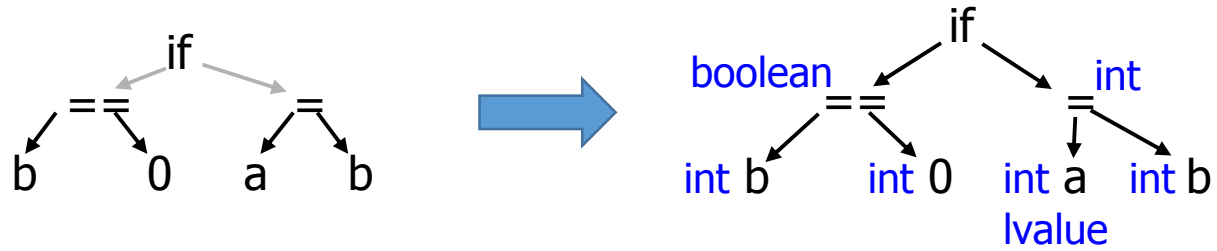
Semantic analysis

Structure of a compiler



Semantic analysis

Input: syntax tree ⇒ **Output:** decorated syntax tree



- Context-free grammar can not represent all language constraints, e.g. non local/context-dependent relations.
- Semantic analysis checks the source program for semantic consistency with the language definition.
  - A variable can not be used without having been defined
  - The same variable/function/class/method can not be defined twice
  - The number of arguments of a function should match its definition
  - One can not multiply a number and a string
  - ...
- Last compiler phase that can report errors to the user.
- Also, figures out useful information for later compiler phases (e.g., types of all expressions)

Outline

1. Syntax-directed translation
2. Abstract syntax tree
3. Type and scope checking

Syntax-directed definition

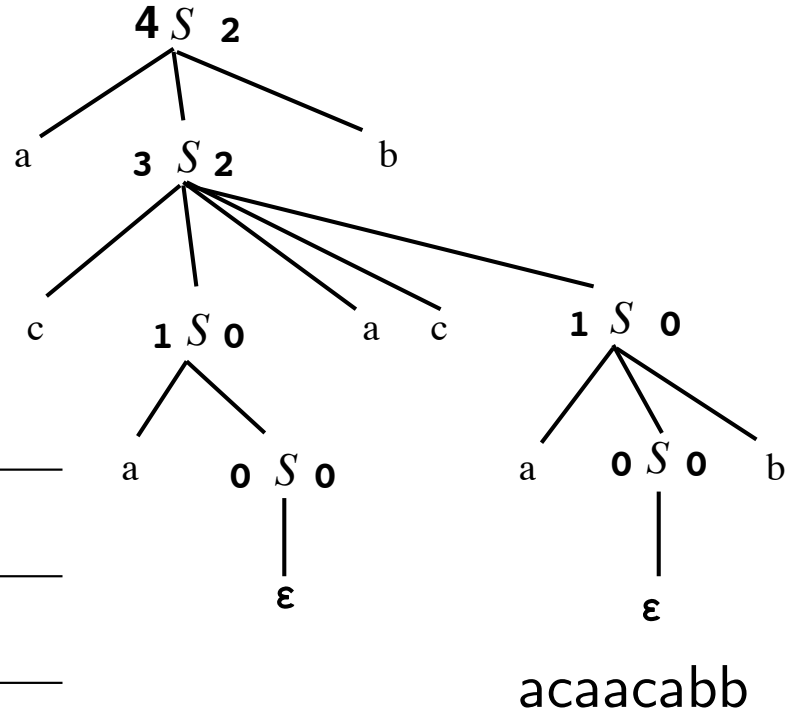
- A general way to associate actions (i.e., programs) to production rules of a context-free grammar
- Used for carrying out most semantic analyses as well as code translation

- A syntax-directed definition associates:
  - With each grammar symbol, a set of attributes, and
  - With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production
- A grammar with attributes and semantic rules is called an attributed grammar
- A parse tree augmented with the attribute values at each node is called an annotated parse tree.

Example

Grammar:

$$S \rightarrow aSb|aS|cSacS|\epsilon$$



Semantic rules:

Production	Semantic rules
$S \rightarrow aS_1b$	$S.nba := S_1.nba + 1$ $S.nbc := S_1.nbc$
$S \rightarrow aS_1$	$S.nba := S_1.nba + 1$ $S.nbc := S_1.nbc$
$S \rightarrow cS_1acS_2$	$S.nba := S_1.nba + S_2.nba + 1$ $S.nbc := S_1.nbc + S_2.nbc + 2$
$S \rightarrow \epsilon$	$S.nba := 0$ $S.nbc := 0$
$S' \rightarrow S$	Final result is in $S.nba$ and $S.nbc$

(subscripts allow to distinguish different instances of the same symbol in a rule)

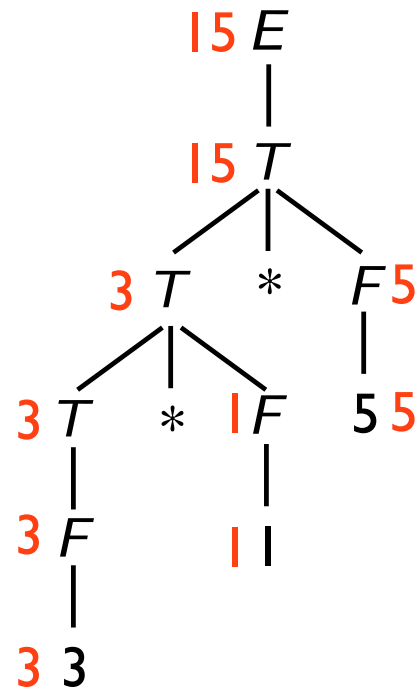
Attributes

- Two kinds of attributes
  - Synthesized: Attribute value for the LHS nonterminal is computed from the attribute values of the symbols at the RHS of the rule.
  - Inherited: Attribute value of a RHS nonterminal is computed from the attribute values of the LHS nonterminal and some other RHS nonterminals.
- Terminals can have synthesized attributes, computed by the lexer (e.g., *id*.*lexeme*), but no inherited attributes.

Example: synthesized attributes to evaluate expressions

Left-recursive expression grammar

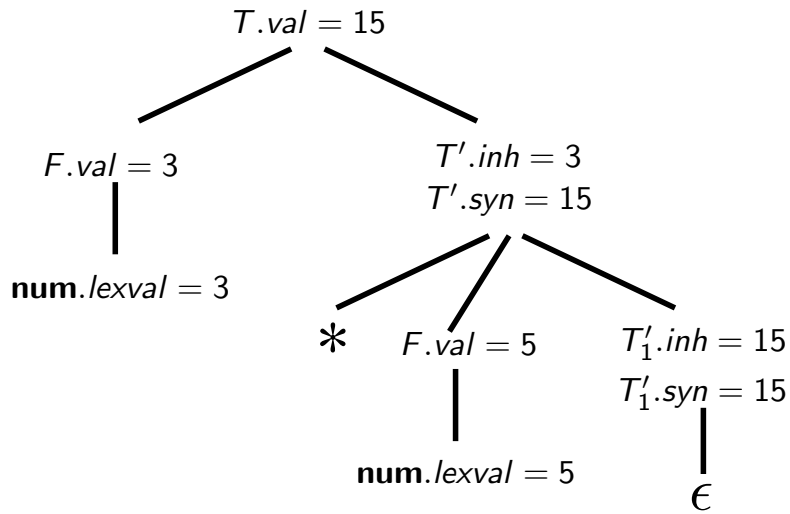
Production	Semantic rules
$L \to E$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to$ **num**	$F.val =$ **num**$.lexval$



Example: inherited attributes to evaluate expressions

LL expression grammar

Production	Semantic rules
$T \to FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
$T' \to *FT'_1$	$T'_1.inh = T'.inh \times F.val$
	$T'.syn = T'_1.syn$
$T' \to \epsilon$	$T'.syn = T'.inh$
$F \to$ **num**	$F.val =$ **num**$.lexval$



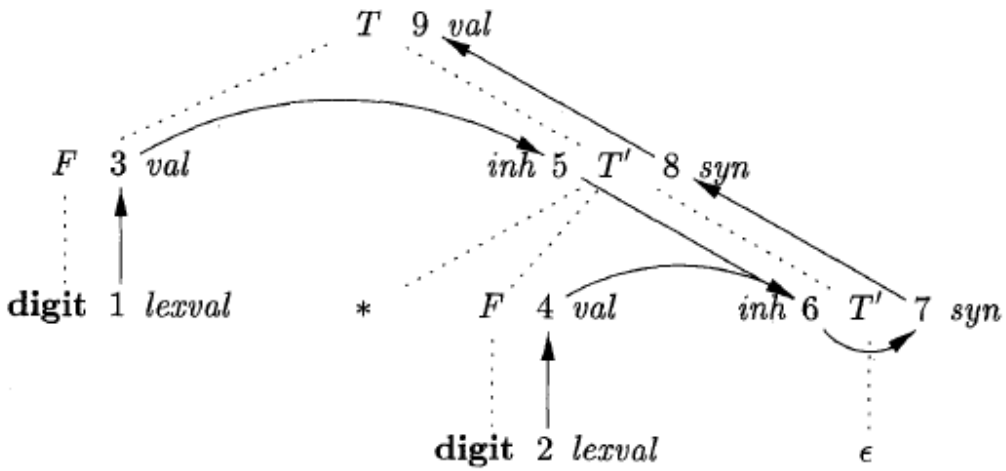
Evaluation order of SDD's

General case of synthesized and inherited attributes:

- Draw a dependency graph between attributes on the parse tree
- Find a topological order on the dependency graph (possible if and only if there are no directed cycles)
- If a topological order exists, it gives a working evaluation order. If not, it is impossible to evaluate the attributes

In practice, it is difficult to predict from an attributed grammar whether no parse tree will have cycles

Example:

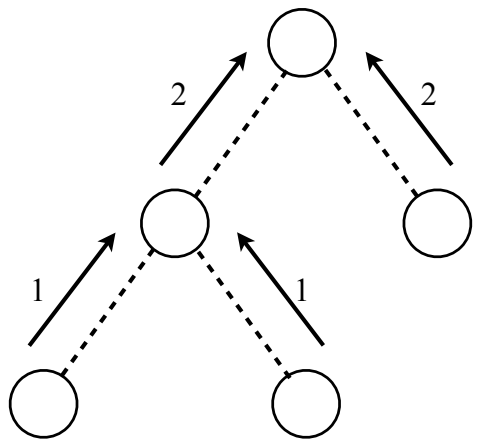


(Dragonbook)

Evaluation order of SDD's

Some important particular cases:

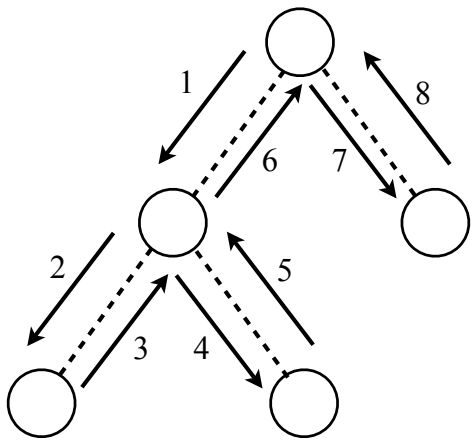
- A grammar with only synthesized attributes is called a *S*-attributed grammar.
- Attributes can be evaluated by a bottom-up (postorder) traversal of the parse tree



Evaluation order of SDD's

Some important particular cases:

- A syntax-directed definition is *L*-attributed if each attribute is either
  1. Synthesized
  2. Inherited "from the left": if the production is $A \to X_1 X_2 \ldots X_n$, then the inherited attributes for X_j can depend only on
     - 2.1 Inherited attributes of A
     - 2.2 Any attributes among $X_1, \ldots, X_{j-1}$ (symbols at the left of X_j
     - 2.3 Attributes of X_j (provided they are not causing cycles)
- To evaluate the attributes: do a depth first traversal evaluating inherited attributes on the way down and synthesized attributes on the way up (i.e., an Euler-tour traversal)



Translation of code

- Syntax-directed definitions can be used to translate code
- Example: translating expressions to post-fix notation

Production	Semantic rules
$L \to E$	$L.t = E.t$
$E \to E_1 + T$	$E.t = E_1.t \Vert T.t \Vert '+'$
$E \to E_1 - T$	$E.t = E_1.t \Vert T.t \Vert '-'$
$E \to T$	$E.t = T.t$
$T \to T_1 * F$	$T.t = T_1.t \Vert F.t \Vert '*'$
$T \to F$	$T.t = F.t$
$F \to (E)$	$F.t = E.t$
$F \to$ **num**	$F.t =$ **num**.*lexval*

Syntax-directed translation scheme

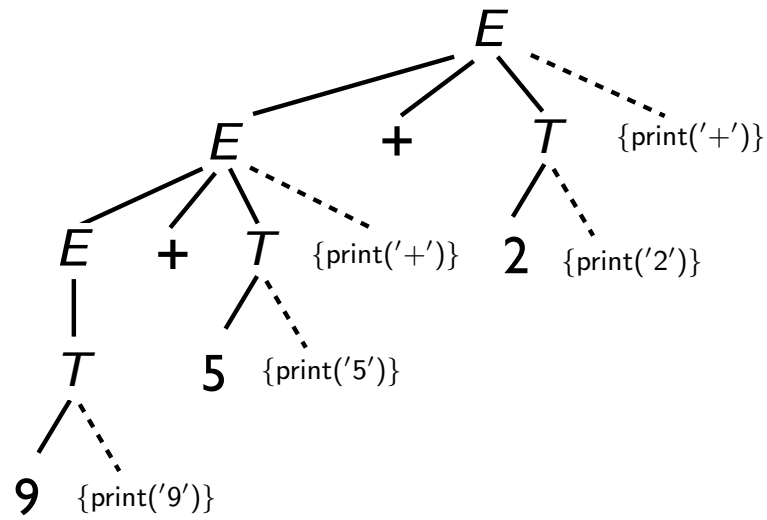
- The previous solution requires to manipulate strings (concatenate, create, store)
- An alternative is to use syntax-directed translation schemes.
- A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments (called semantic actions) embedded within production bodies:

$$A \to \{R_0\} X_1 \{R_1\} X_2 \ldots X_k \{R_k\}$$

- Actions are performed from left-to-right when the rules is used for a reduction
- Interesting for example to generate code incrementally

Example for code translation

Production	
$L \to E$	
$E \to E_1 + T$	$\{\text{print}('+')\}$
$E \to T$	
$T \to T_1 * F$	$\{\text{print}('*')\}$
$T \to F$	
$F \to (E)$	
$F \to$ **num**	$\{\text{print}(\textbf{num}.lexval)\}$



(Post-fix SDT as all actions are performed at the end of the productions)

Side-effects

- Semantic rules and actions in SDD and SDT's can have side-effects. E.g., for printing values or adding information into a table
- Needs to ensure that the evaluation order is compatible with side-effects
- Example: variable declaration in C

Production	Semantic rules	
$D \to TL$	$L.type = T.type$	(inherited)
$T \to$ int	$T.type =$int	(synthesized)
$T \to$ float	$T.type =$float	(synthesized)
$L \to L_1$, id	$L_1.type = L.type$	(inherited)
	$AddType(\mathbf{id}.entry, L.type)$	(synthesized, side effect)
$L \to \mathbf{id}$	$AddType(\mathbf{id}.entry, L.type)$	(synthesized, side effect)

- **id**.*entry* is an entry in the symbol table. *AddType* adds type information about entry in the symbol table

Implementation of SDD's: after parsing

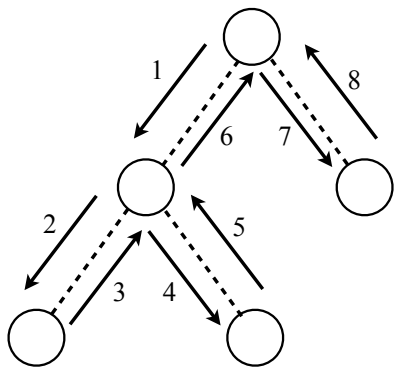
Attributes can be computed after parsing:

- By explicitely traversing the parse or syntax tree in any order permitting the evaluation of the attributes
- Depth-first for S-attributed grammars or Euler tour for L-attributed grammar
- Advantage: does not depend on the order imposed by the syntax analysis
- Drawback: requires to build (and store in memory) the syntax tree

Evaluation after parsing of *L*-attributed grammar

For *L*-attributed grammars, the following recursive function will do the computation for inherited and synthesized attributes

```
ANALYSE(N, InheritedAttributes)
    if LEAF(N)
        return SynthesizedAttributes
    Attributes = InheritedAttributes
    for each child C of N, from left to right
        ChildAttributes = ANALYSE(C, Attributes)
        Attributes = Attributes ∪ ChildAttributes
    Execute semantic rules for the production at node N
    return SynthesizedAttributes
```



- Inherited attributes are passed as arguments and synthesized attributes are returned by recursive calls
- In practice, this is implemented as a big two-level switch on nonterminals and then rules with this nonterminal at its LHS

Variations

- Instead of a giant switch, one could have separate routines for each nonterminal (as with recursive top-down parsing) and a switch on productions having this nonterminal as LHS (see examples later)
- Global variables can be used instead of parameters to pass inherited attributes by side-effects (with care)
- Can be easily adapted to use syntax-directed translation schemes (by interleaving child analysis and semantic actions)

- In object-oriented languages, this can be implemented with:
  - one class for each nonterminal symbol, or for each syntactic category (expression, statement, etc.)
  - either one 'Analyse' method in each class (difficult to maintain, extend) or using the visitor design pattern.

Implementation of SDD's: during parsing

Attributes can be computed directly during parsing:

- Attributes of a S-attributed grammar are easily computed during bottom-up parsing
- Attributes of a L-attributed grammar are easily computed during top-down parsing
- Attribute values can be stored on a stack (the same as the one for parsing or a different one)
- Advantage: one pass, does not require to store (or build) the syntax tree, memory efficient
- Drawback: the order of evaluation is constrained by the parser, not modular

Nowadays, mostly only used to generate the abstract syntax tree

Bottom-up parsing and S-attributed grammar

- Synthesized attributes are easily handled during bottom-up parsing. Handling inherited attributes is possible (for a LL-grammar) but more difficult.
- Example with only synthesized attributes (stored on a stack):

Production	Semantic rules	Stack actions
$E \to E_1 + T$	$E.val = E_1.val + T.val$	$tmpT = \text{POP}()$
		$tmpE = \text{POP}()$
		$\text{PUSH}(tmpE + tempT)$
$E \to T$	$E.val = T.val$	
$T \to T_1 * F$	$T.val = T_1.val \times F.val$	$tmpT = \text{POP}()$
		$tmpF = \text{POP}()$
		$\text{PUSH}(tmpT * tempF)$
$T \to F$	$T.val = F.val$	
$F \to (E)$	$F.val = E.val$	
$F \to$ **num**	$F.val =$ **num**$.lexval$	$\text{PUSH}($**num**$.lexval)$

(Parsing tables on slide 189)

Bottom-up parsing and S-attributed grammar

Stack	Input	Action	Attribute stack
$ 0	$2 * (10 + 3)$	s5	
$ 0 2 5	$*(10 + 3)$	r6: $F \rightarrow$ **num**	2
$ 0 F 3	$*(10 + 3)$	r4: $T \rightarrow F$	2
$ 0 T 2	$*(10 + 3)$	s7	2
$ 0 T 2 * 7	$(10 + 3)$	s4	2
$ 0 T 2 * 7 (4	$10 + 3)$	s5	2
$ 0 T 2 * 7 (4 10 5	$+3)$	r6: $F \rightarrow$ **num**	2 10
$ 0 T 2 * 7 (4 F 3	$+3)$	r4: $T \rightarrow F$	2 10
$ 0 T 2 * 7 (4 T 2	$+3)$	r2: $E \rightarrow T$	2 10
$ 0 T 2 * 7 (4 E 8	$+3)$	s6	2 10
$ 0 T 2 * 7 (4 E 8 + 6	$3)$	s5	2 10
$ 0 T 2 * 7 (4 E 8 + 6 3 5	$)$	r6: $F \rightarrow$ **num**	2 10 3
$ 0 T 2 * 7 (4 E 8 + 6 F 3	$)$	r4: $T \rightarrow F$	2 10 3
$ 0 T 2 * 7 (4 E 8 + 6 T 9	$)$	r1: $E \rightarrow E + T$	2 13
$ 0 T 2 * 7 (4 E 8	$)$	s11	2 13
$ 0 T 2 * 7 (4 E 8) 11	$	r5: $F \rightarrow (E)$	2 13
$ 0 T 2 * 7 F 10	$	r3: $T \rightarrow T * F$	26
$ 0 T 2	$	r2: $E \rightarrow T$	26
$ 0 E 1	$	Accept	26

Top-down parsing of L-attributed grammar

- Recursive parser: the analysis scheme of slide 248 can be incorporated within the recursive functions of nonterminals
- Table-driven parser: this is also possible but less obvious.
- Example with only inherited attributes (stored on a stack):

Production	Semantic rules	Stack actions
$S' \to S$	$S.nb = 0$	$\textsc{push}(0)$
$S \to (S_1)S_2$	$S_1.nb = S.nb + 1$ $S_2.nb = S.nb$	$\textsc{push}(\textsc{top}() + 1)$
$S \to \epsilon$	$\textsc{print}(S.nb)$	$\textsc{print}(\textsc{pop}())$

(print the depth of nested parentheses)

Parsing table:

	(	)	\$
S'	$S' \to S$		$S' \to S$
S	$S \to (S)S$	$S \to \epsilon$	$S \to \epsilon$

Top-down parsing of L-attributed grammar

Stack	Input	Attribute stack	Output
$S'\$$	(()(()))()	0	
$S\$$	(()(()))()	0 1	
$(S)S\$$	(()(()))()	0 1	
$S)S\$$	()(()))()	0 1 2	
$(S)S)S\$$	()(()))()	0 1 2	
$S)S)S\$$	)(()))()	0 1	2
$)S)S\$$	)(()))()	0 1	
$S)S\$$	(()))()	0 1 2	
$(S)S)S\$$	(()))()	0 1 2	
$S)S)S\$$	()))()	0 1 2 3	
$(S)S)S)S\$$	()))()	0 1 2 3	
$S)S)S)S\$$	)))()	0 1 2	3
$)S)S)S\$$	)))()	0 1 2	
$S)S)S\$$	))()	0 1	2
$)S)S\$$	))()	0 1	
$S)S\$$	)()	0	1
$)S\$$	)()	0	
$S\$$	()	0 1	
$(S)S\$$	()	0 1	
$S)S\$$	)	0	1
$)S\$$	)	0	
$S\$$			0
$\$$			

Comments

- It is possible to transform a grammar with synthesized and inherited attributes into a grammar with only synthesized attributes
- It is usually easier to define semantic rules/actions on the original (ambiguous) grammar, rather than the transformed one
- There are techniques to transform a grammar with semantic actions (see reference books for details)

Applications of SDD's

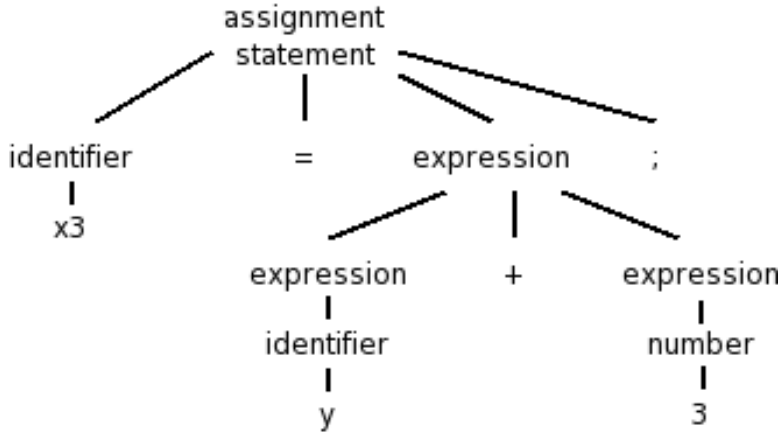
SDD can be used at several places during compilation:

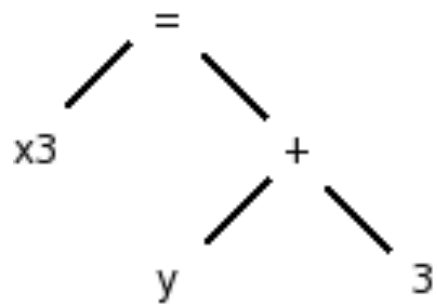
- Building the syntax tree from the parse tree
- Various static semantic checking (type, scope, etc.)
- Code generation
- Building an interpreter
- ...

Outline

1. Syntax-directed translation
2. Abstract syntax tree
3. Type and scope checking

Abstract syntax tree (AST)



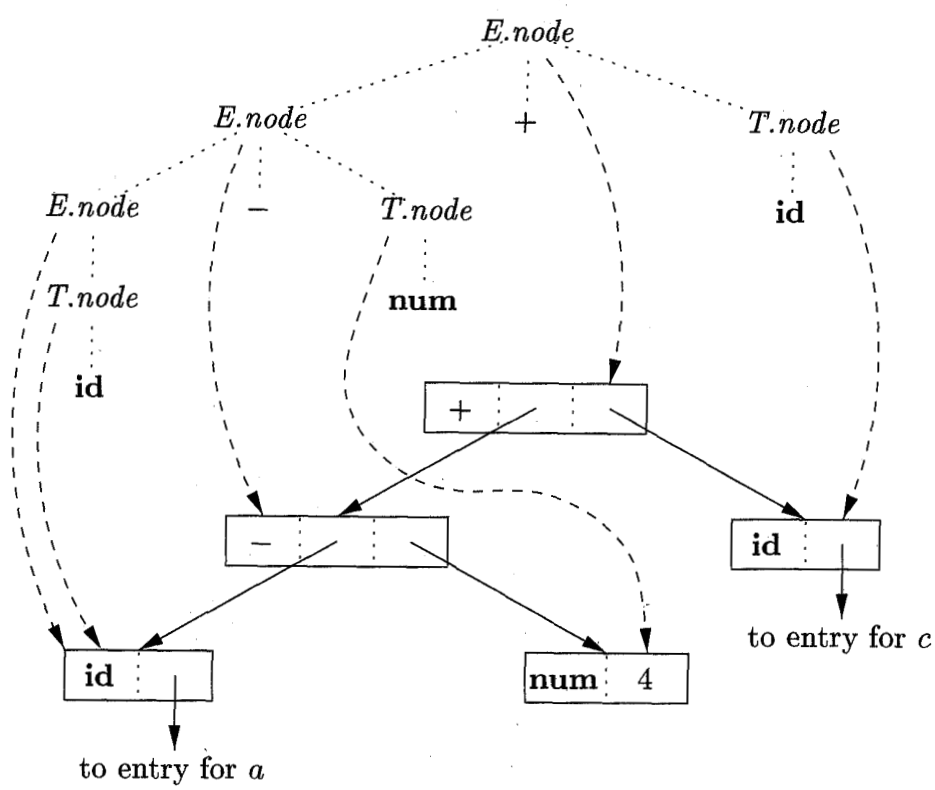


- The abstract syntax tree is used as a basis for most semantic analyses and for intermediate code generation (or even used as an intermediate representation)
- When the grammar has been modified for parsing, the syntax tree is a more natural representation than the parse tree
- The abstract syntax tree can be constructed using SDD (see next slides)
- Another SDD can then be defined on the syntax tree to perform semantic checking or generate another intermediate code (directed by the syntax tree and not the parse tree)

Generating an abstract syntax tree

For the left-recursive expression grammar:

Production	Semantic rules
$E \to E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
$E \to E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
$E \to T$	$E.node = T.node$
$T \to (E)$	$T.node = E.node$
$T \to$ **id**	$T.node =$ **new** $Leaf($**id**, **id**$.entry)$
$T \to$ **num**	$T.node =$ **new** $Leaf($**num**, **num**$.entry)$

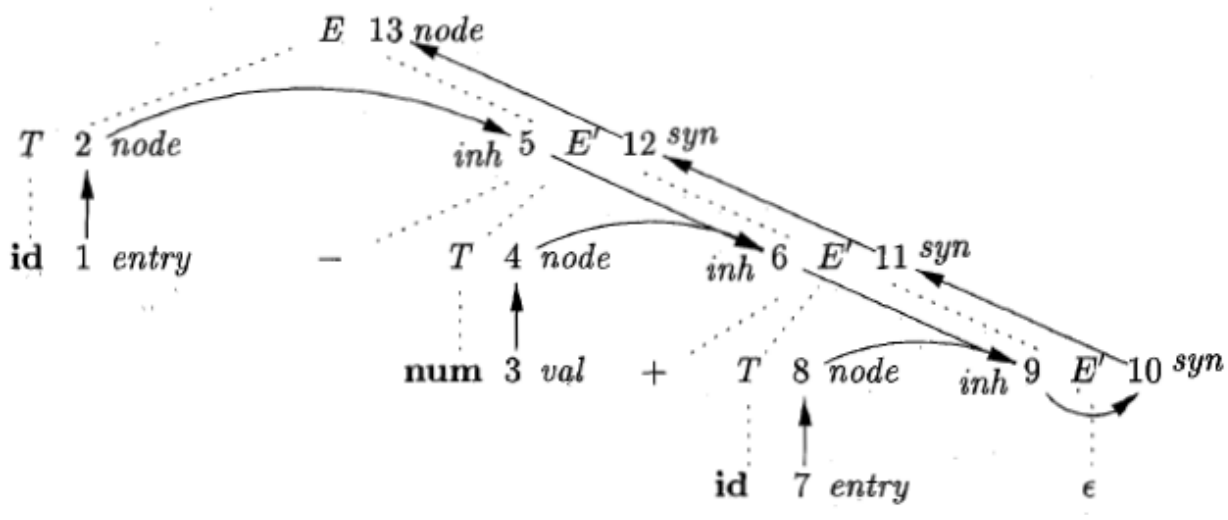


(Dragonbook)

Generating an abstract syntax tree

For the LL transformed expression grammar:

Production	Semantic rules
$E \to TE'$	$E.node = E'.syn;\ E'.inh = T.node$
$E' \to +TE'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node);\ E'.syn = E'_1.syn$
$E' \to -TE'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node);\ E'.syn = E'_1.syn$
$E' \to \epsilon$	$E'.syn = E'.inh$
$E \to T$	$E.node = T.node$
$T \to (E)$	$T.node = E.node$
$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$



(Dragonbook)

Syntactic sugar

- Syntactic sugar: syntax within a programming language that makes things easier to read or to express but does not affect functionality and expressive power of the language.
- When building the syntax tree, it is useful to remove sugared constructs (= "desugaring").
- It makes subsequent phases easier to implement and maintain.
- Examples:
  - In C: `a[i]` (= `(a+i)*`), `a->x` (= `(*a).x`)
  - All loop operators (`for`, `repeat`, `while`) can be rewritten as while loops.
- Information should however be kept about original code for semantic error reporting.

Outline

1. Syntax-directed translation
2. Abstract syntax tree
3. Type and scope checking

Type and scope checking

- Static checkings:
  - All checkings done at compilation time (versus dynamic checkings done at run time)
  - Allow to catch errors as soon as possible and ensure that the program can be compiled
- Two important checkings:
  - Scope checking: checks that all variables and functions used within a given scope have been correctly declared
  - Type checking: ensures that an operator or function is applied to the correct number of arguments of the correct types
- These two checks are based on information stored in a symbol table

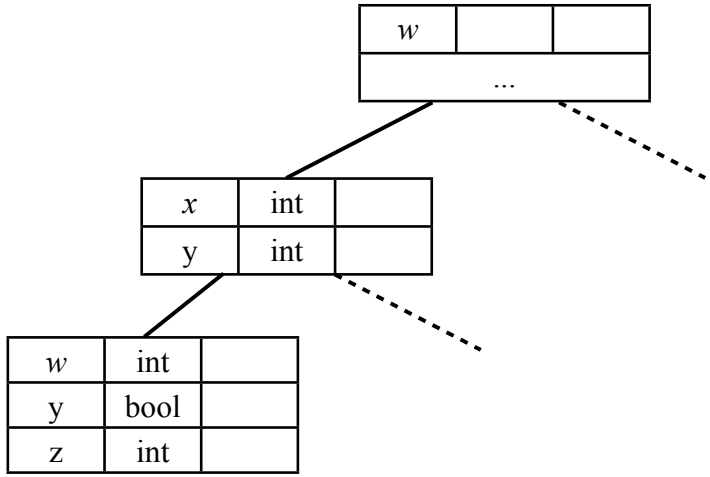
Scope

```
{
   int x = 1;
   int y = 2;
   {
      double x = 3.1416;
      y += (int)x;
   }
   y += x;
}
```

- Most languages offer some sort of control for scopes, constraining the visibility of an identifier to some subsection of the program
- A scope is typically a section of program text enclosed by basic program delimiters, e.g., {} in C, begin-end in Pascal.
- Many languages allow nested scopes, i.e., scopes within scopes. The current scope (at some program position) is the innermost scope.
- Global variables and functions are available everywhere
- Determining if an identifier encountered in a program is accessible at that point is called Scope checking.

Symbol table

```
{ int x; int y;
   {  int w; bool y; int z;
       ..w..; ..x..; ..y..; ..z..;
   }
   ..w..; ..x..; ..y..;
}
```



- The compiler keeps track of names and their binding using a symbol table (also called an environment)
- A symbol table must implement the following operations:
  - Create an empty table
  - Add a binding between a name and some information
  - Look up a name and retrieve its information
  - Enter a new scope
  - Exit a scope (and reestablish the symbol table in its state before entering the scope)

Symbol table

- To manage scopes, one can use a persistent or an imperative data structure
- A persistent data structure is a data structure which always preserves the previous version of itself when it is modified
- Example: lists in functional languages such as Scheme
  - Binding: insert the binding at the front of the list, lookup: search the list from head to tail
  - Entering a scope: save the current list, exiting: recalling the old list
- A non persistent implementation: with a stack
  - Binding: push the binding on top of the stack, lookup: search the stack from top to bottom
  - Entering a scope: push a marker on the top of the stack, exiting: pop all bindings from the stack until a marker is found, which is also popped
  - This approach destroys the symbol table when exiting the scope (problematic in some cases)

More efficient data structures

- Search in list or stack is $O(n)$ for n symbols in the table
- One can use more efficient data structures like hash-tables or binary search trees
- Scopes can then be handled in several ways:
  - Create a new symbol table for each scope and use a stack or a linked list to link them
  - Use one big symbol table for all scopes:
    - Each scope receives a number
    - All variables defined within a scope are stored with their scope number
    - Exiting a scope: removing all variables with the current scope number
  - There exist persistent hash-tables

Types

- Type checking is verifying that each operation executed in a program respects the type system of the language, i.e., that all operands in any expression are of appropriate types and number
- Type systems:
  - Static typing if checking is done at compilation-time (e.g., C, Java, C++)
  - Dynamic typing if checking is done at run-time (e.g., Scheme, Javascript).
  - Strong typing if all type errors are caught (e.g., Java, Scheme)
  - Weak typing if operations may be performed on values of wrong types (e.g., C, assembly)
- Implicit type conversion, or coercion, is when a compiler finds a type error and changes the type of the variable into the appropriate one (e.g., integer→float)

Principle of static type checking

- Identify the types of the language and the language constructs that have types associated with them
- Associate a type attribute to these constructs and semantic rules to compute them and to check that the typing system is respected
- Needs to store identifier types in the symbol table
- One can use two separate tables, one for the variable names and one for the function names
- Function types is determined by the types (and number) of arguments and return type. E.g., $(int, int) \rightarrow int$
- Type checking can not be dissociated from scope and other semantic checking

Illustration

We will use the following source grammar to illustrate type and scope checking

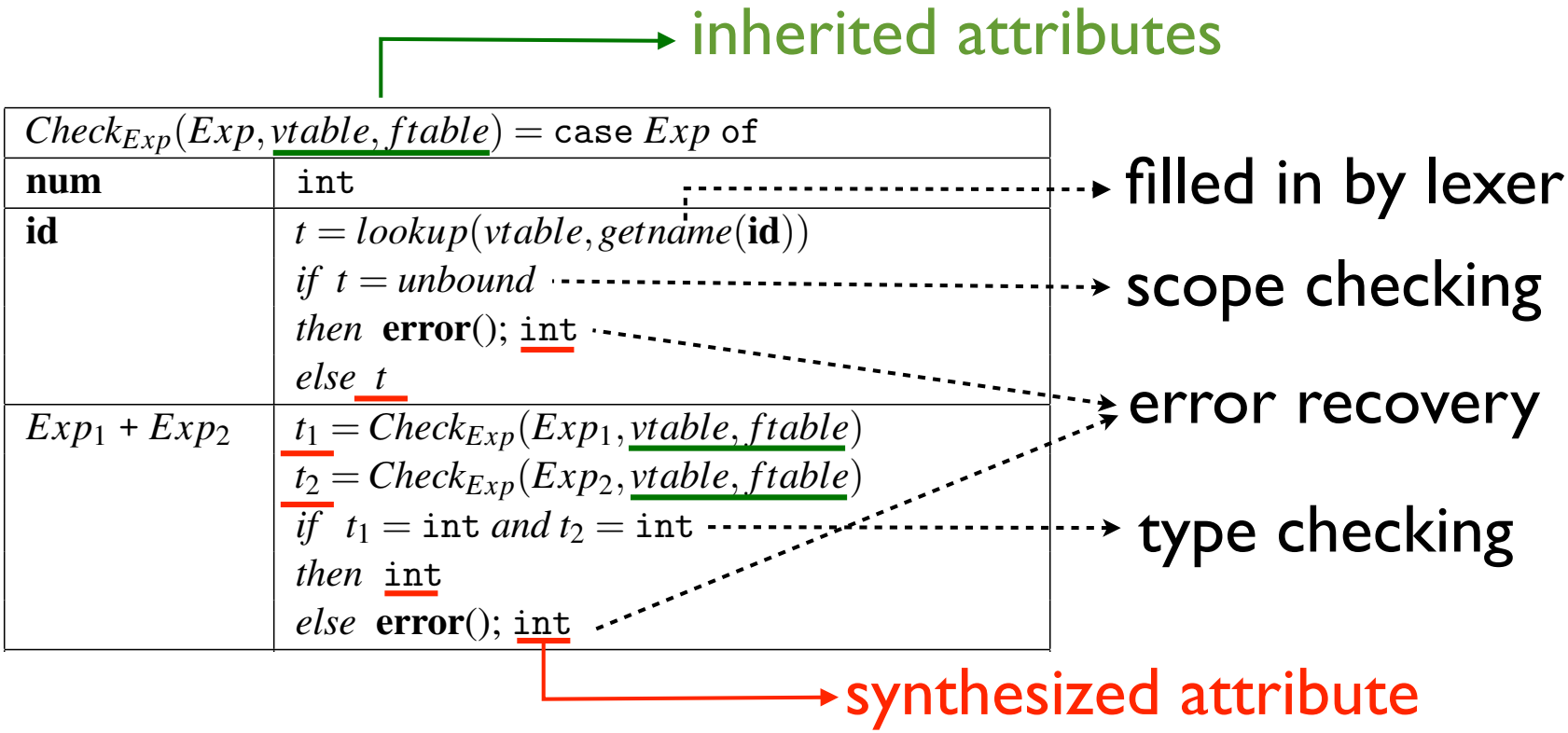
$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Funs} \\
\\
\mathit{Funs} & \rightarrow & \mathit{Fun} \\
\mathit{Funs} & \rightarrow & \mathit{Fun}\ \mathit{Funs} \\
\\
\mathit{Fun} & \rightarrow & \mathit{TypeId}\ (\ \mathit{TypeIds}\ )\ =\ \mathit{Exp} \\
\\
\mathit{TypeId} & \rightarrow & \texttt{int}\ \mathbf{id} \\
\mathit{TypeId} & \rightarrow & \texttt{bool}\ \mathbf{id} \\
\\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId} \\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId}\ ,\ \mathit{TypeIds}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \mathbf{num} \\
\mathit{Exp} & \rightarrow & \mathbf{id} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} = \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathbf{id}\ (\ \mathit{Exps}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{let}\ \mathbf{id} = \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \\
\\
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ ,\ \mathit{Exps}
\end{array}
$$

(see chapter 5 and 6 of (Mogensen, 2010) for full details)

Implementation on the syntax tree: expressions

Type checking of expressions:



Follows the implementation of slide 249 with one function per nonterminal, with a switch on production rules. Could be implemented with classes/methods or a visitor pattern.

Implementation on the syntax tree: function calls

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
id ($Exps$)	$t = lookup(ftable, getname(\textbf{id}))$ *if* $t = unbound$ *then* **error**(); `int` *else* $((t_1, \ldots, t_n) \rightarrow t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ *if* $m = n$ *and* $t_1 = t'_1, \ldots, t_n = t'_n$ *then* t_0 *else* **error**(); t_0

filled in by lexer

scope checking

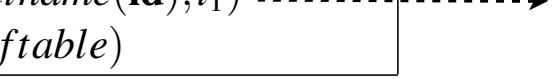
checking function arguments

$Check_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$[Check_{Exp}(Exp, vtable, ftable)]$
Exp , $Exps$	$Check_{Exp}(Exp, vtable, ftable)$ :: $Check_{Exps}(Exps, vtable, ftable)$

$\approx$ cons

Implementation on the syntax tree: variable declaration

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
`let` **id** = Exp_1 `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$



- Create a new symbol table $vtable'$ with the new binding
- Pass it as an argument for the evaluation of Exp_2 (*right* child)

Implementation on the syntax tree: function declaration

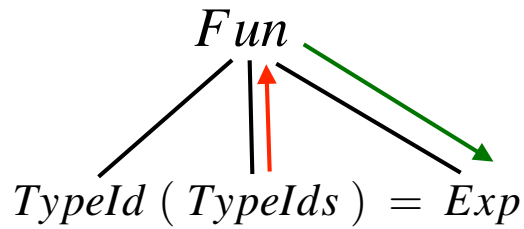
synthesized attribute

$Check_{Fun}(Fun, ftable) =$ `case` Fun `of`	
$TypeId$ ($TypeIds$) = Exp	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ *if* $t_0 \neq t_1$ *then* **error**()

inherited attributes

$Get_{TypeId}(TypeId) =$ `case` $TypeId$ `of`	
`int` **id**	$(getname(\textbf{id}),$ `int`$)$
`bool` **id**	$(getname(\textbf{id}),$ `bool`$)$

$Check_{TypeIds}(TypeIds) =$ `case` $TypeIds$ `of`	
$TypeId$	$(x,t) = Get_{TypeId}(TypeId)$ $bind(emptytable, x, t)$
$TypeId$, $TypeIds$	$(x,t) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ *if* $lookup(vtable, x) = unbound$ *then* $bind(vtable, x, t)$ *else* **error**(); $vtable$



Create a symbol table with arguments

Implementation on the syntax tree: program

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$ $if\ lookup(ftable, \texttt{main}) \neq (\texttt{int}) \rightarrow \texttt{int}$ $then\ \textbf{error}()$

Collect all function definitions in a symbol table (to allow mutual recursion)

Language semantic requires a main function

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
$Fun\ Funs$	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

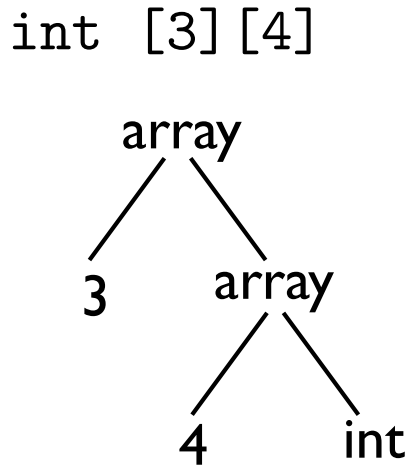
- Needs two passes over the function definitions to allow mutual recursion
- See (Mogensen, 2010) for Get_{Funs} (similar to $Check_{Funs}$)

More on types

Compound types:

- They are represented by trees (constructed by a SDD)
- Example: array declarations in C

Production	Semantic rules
$T \rightarrow BC$	$T.t = C.t$; $C.b = B.t$
$B \rightarrow$ int	$B.t =$int
$B \rightarrow$ float	$B.t =$float
$C \rightarrow$ [NUM]C_1	$C.t = array($NUM$.val, C_1.t)$
$C \rightarrow \epsilon$	$C.t = C.b$



- Compound types are compared by comparing their trees

More on types

Type coercion:

- The compiler supplies implicit conversions of types
- Define a hierarchy of types and convert each operand to their least upper bound (LUB) in the hierarchy

Overloading:

- The same name is used for several different operations over several different types (e.g., = in our source language)
- Type must be defined at translation

$Exp_1 = Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ *if* $t_1 = t_2$ *then* `bool` *else* **error**(); `bool`

Polymorphism/generic types:

- Functions defined over a large class of similar types
- E.g: Functions that can be applied over all arrays no matter what the types of the elements are

More on types

Dynamic typing:

- Type checking is (mostly) done at run-time
- Objects (values) have types, not variables
- Dynamically typed languages: Scheme, Lisp, Python, Php...
- The following scheme code will generate an error at run time

```
(defun length
    (lambda (l)
        (if (null? l)
            0
            (+ 1 (length (cdr l))))))
(length 4)
```

More on types

Implicit types and type inference:

- Some languages (like ML, (O)Caml, Haskell, C#) do not require to explicitly declare types of (all) functions or variables
- Types are automatically inferred at compile time.
- The following OCaml code will generate an error at compile time

```
let rec length = function
[] -> 0
| h::t -> 1 + (length t);;

let myf () = length 4;;
```

- In our illustrative example, there is inference in the let expression:

`let` **id** = Exp_1 `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\mathbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

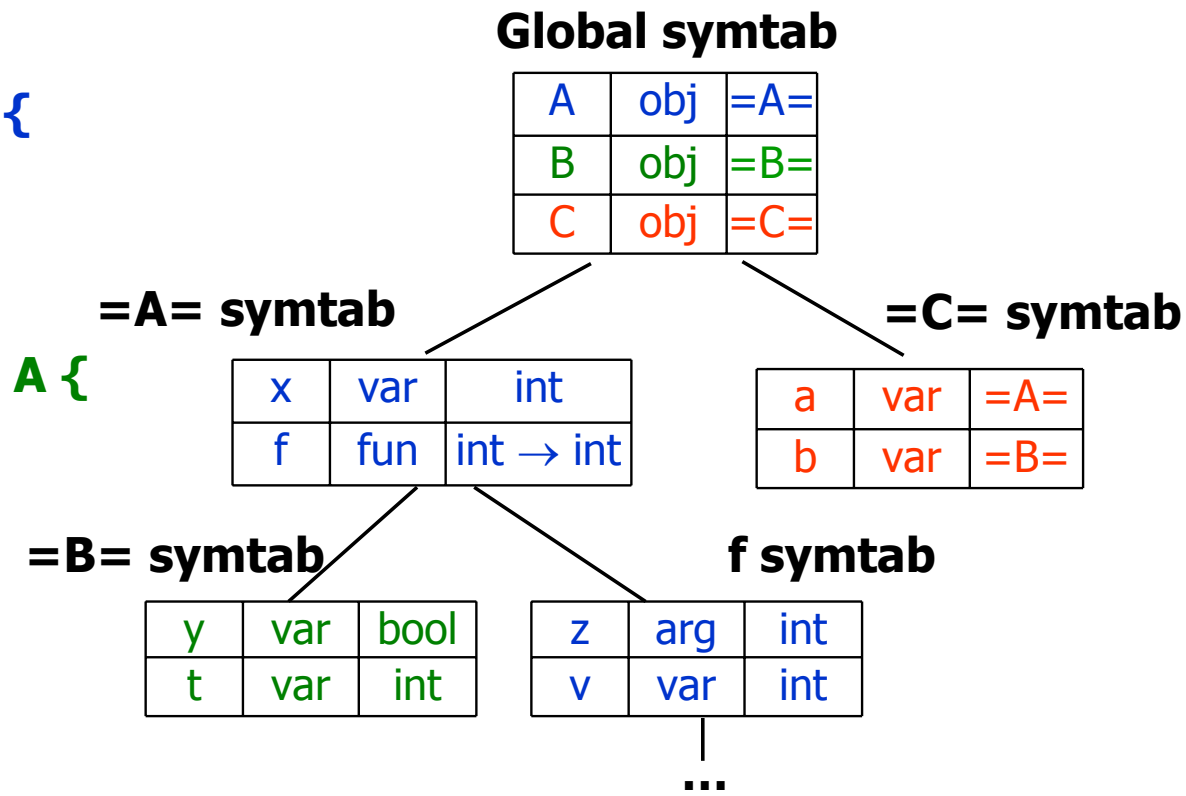
The type of **id** is inferred from the type of Exp_1.

Scope and type checking for object-oriented languages

- Each class declaration is added to the (global) symbol table or to a separate table for types
- Each class declaration also introduces a new scope:
  - that contains all declared fields and methods
  - that have scopes of methods as sub-scopes
- Inheritance implies a hierarchy of class scopes
  - If class B extends class A, then scope of A is a parent scope for B.
- Each scope can be implemented as a single symbol table, combining fields and methods, or as two separate tables, one for fields and one for methods.

Example

```
class A {
    int x;
int f(int z) {
    int v; ...
    }
}

class B extends A {
    bool y;
    int t;
}

class C {
    A a;
    B b;
    ...
a = b;
}
```



(Teitelbaum)

Scope checking

Resolve identifier occurrence in a method:

- Start the search from the symbol table of the current block and then walk the symbol table hierarchy

Resolve qualified accesses: e.g., $o.f$, where o is of class A.

- Walk the symbol table hierarchy starting with the symbol table of class A.
- `this` (or `self`) keyword starts the walk from the symbol table of the enclosing class.

Note: multiple inheritance (`Class B extends A,C`) can cause problems when the same symbol exist in two parent scopes.

Subtyping

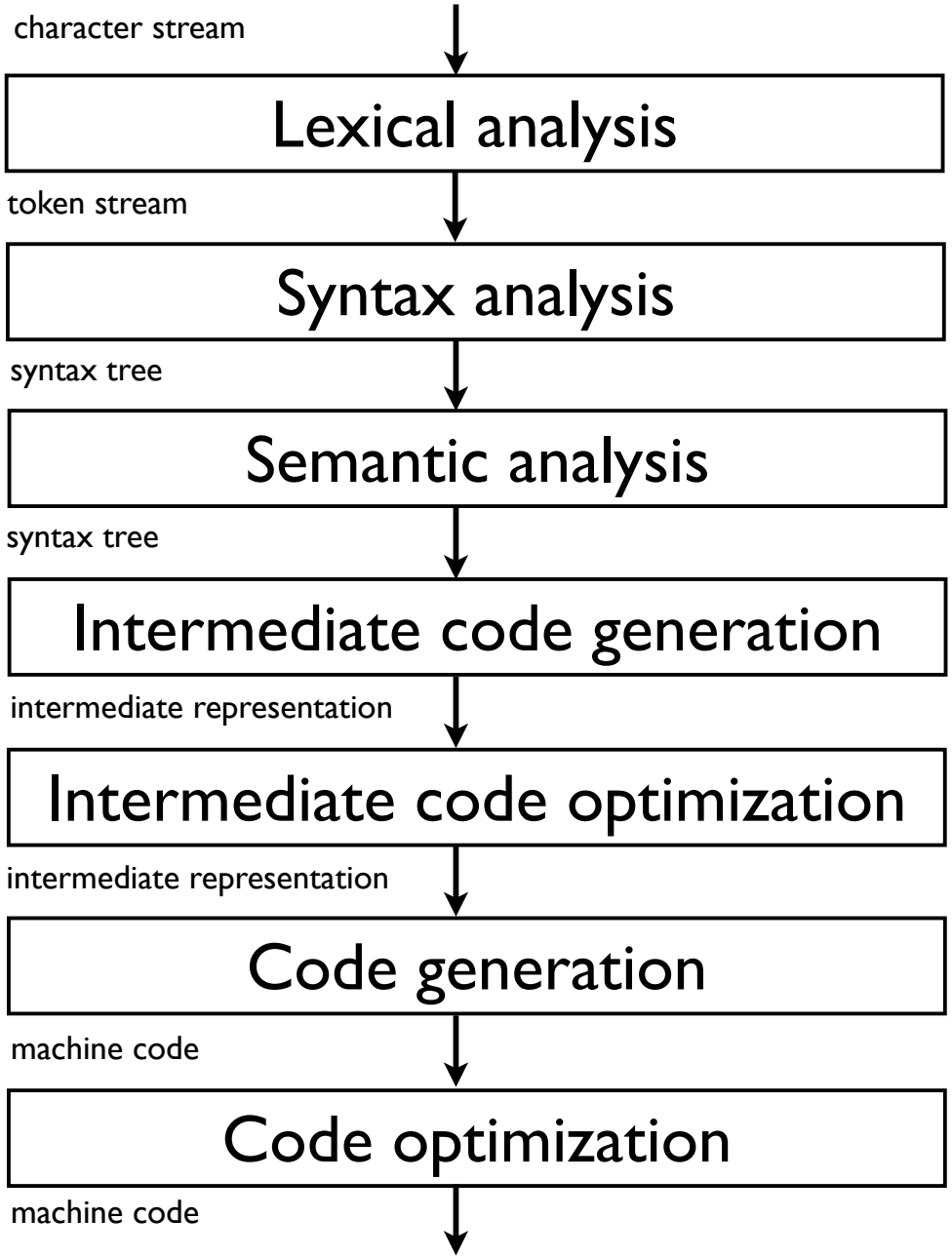
- If class B extends class A, then type B is a subtype of A (type A is a supertype of B).
- Subtype polymorphism: Code using class A objects can also use class B objects.
- For type checking, subtype relations can be tested from the hierarchy of class symbol tables.

- Problems with subtyping:
  - The actual type of an object is unknown at compile time: it can be the declared class or any subclass.
  - Problematic for overridden fields and methods: we don't know which declaration to use at compile time.
  - Solution is language dependent: this can be addressed statically by imposing constraints on language and/or dynamically

Part 5

Intermediate code generation

Structure of a compiler



Outline

1. Intermediate representations
2. Illustration
3. Optimization

Intermediate code generation

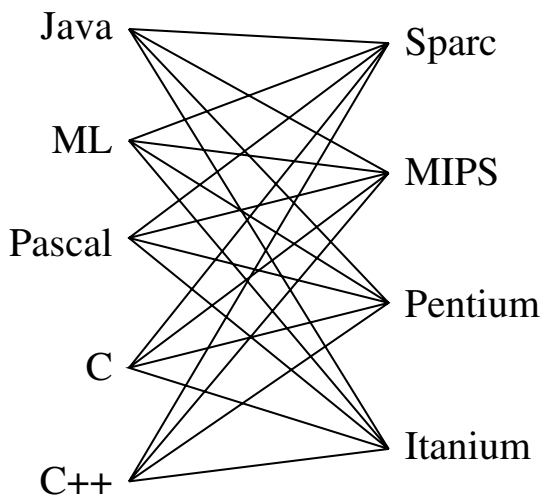
- The final phase of the compiler front-end
- Goal: translate the program into a format expected by the compiler back-end
- In typical compilers: followed by intermediate code optimization and machine code generation

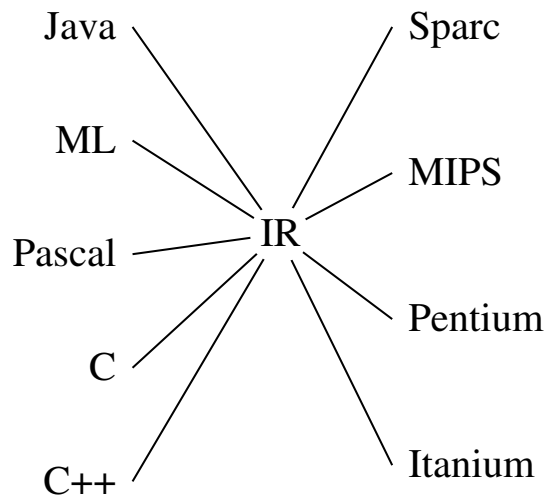
- Techniques for intermediate code generation can be used for final code generation

Intermediate representations

Why use an intermediate representation?

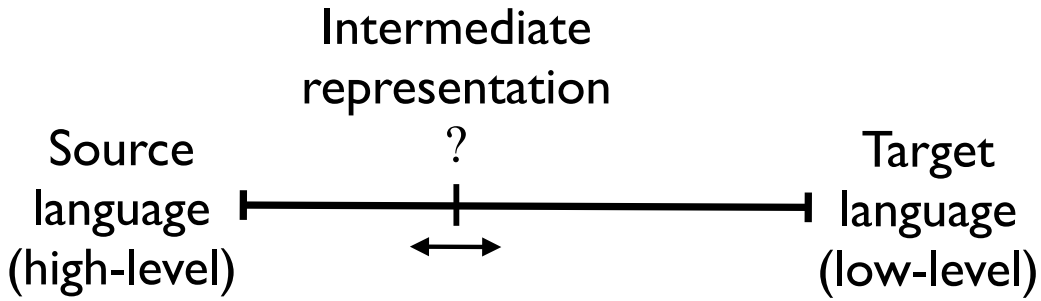
- It's easy to change the source or the target language by adapting only the front-end or back-end (portability)
- It makes optimization easier: one needs to write optimization methods only for the intermediate representation
- The intermediate representation can be directly interpreted





(Appel)

Intermediate representations



- How to choose the intermediate representation?
  - It should be easy to translate the source language to the intermediate representation
  - It should be easy to translate the intermediate representation to the machine code
  - The intermediate representation should be suitable for optimization
- It should be neither too high level nor too low level
- One can have more than one intermediate representation in a single compiler

Some common intermediate representations

General forms of intermediate representations (IR):

- Graphical IR (parse tree, abstract syntax trees, DAG. . .)
- Linear IR (ie., non graphical)
- Three Address Code (TAC): instructions of the form "result=op1 operator op2"
- Static single assignment (SSA) form: each variable is assigned once
- Continuation-passing style (CPS): general form of IR for functional languages

Some common intermediate representations

Examples:

- Java bytecode (executed on the Java Virtual Machine)
- LLVM (Low Level Virtual Machine): SSA and TAC based
- C is used in several compilers as an intermediate representation (Lisp, Haskell, Cython...)
- Microsoft's Common Intermediate Language (CIL)
- GNU Compiler Collection (GCC) uses several intermediate representations:
  - Abstract syntax trees
  - GENERIC (tree-based)
  - GIMPLE (SSA form)
  - Register Transfer Language (RTL, inspired by lisp lists)

(Google them)

Static Single-Assignment Form (SSA)

- A naming discipline used to explicitely encode information about both the flow of control and the flow of data values
- A program is in SSA form if:
  1. each definition has a distinct name
  2. each use refers to a single definition
- Example:

Original code	SSA form
$y = 1$	$y_1 = 1$
$y = 2$	$y_2 = 2$
$x = y$	$x_1 = y_2$

- Main interest: allows to implement several code optimizations.
  - In the example above, it is clear from the SSA form that the first assignment is not necessary.

Converting to SSA

- Converting a program into a SSA form is not a trivial task

Original code

$x = 5$
$x = x - 3$
if $x < 3$
    $y = x * 2$
    $w = y$
else
    $y = x - 3$
$w = x - y$
$z = x + y$

SSA form

$x_1 = 5$
$x_2 = x_1 - 3$
if $x_2 < 3$
    $y_1 = x_2 * 2$
    $w_1 = y_1$
else
    $y_2 = x_2 - 3$
$w_2 = x_2 - y_?$
$z_1 = x_2 + y_?$

- Need to introduce a special statement: Φ-functions

Converting to SSA

Original code

$x = 5$
$x = x - 3$
if $x < 3$
    $y = x * 2$
    $w = y$
else
    $y = x - 3$
$w = x - y$
$z = x + y$

SSA form

$x_1 = 5$
$x_2 = x_1 - 3$
if $x_2 < 3$
    $y_1 = x_2 * 2$
    $w_1 = y_1$
else
    $y_2 = x_2 - 3$
$y_3 = \Phi(y_1, y_2)$
$w_2 = x_2 - y_3$
$z_1 = x_2 + y_3$

- $\Phi(y_1, y_2)$ is defined as y_1 if we arrive at this instruction through the THEN branch, y_2 if through the ELSE branch.
- One needs to introduce Φ functions at every point of the program where several "paths" merge.

SSA form

- Given an arbitrary program, finding where to place the Φ functions is a difficult task.
- However, an efficient solution is available, based on the control flow graph of the program (see later).

- In practice, the Φ functions are not implemented. They indicate to the compiler that the variables given as arguments need to be stored in the same place.
- In the previous example, we can infer that y_1 and y_2 should be stored in the same place

Continuation-passing style (CPS)

- A programming style in functional languages where control is passed explicitely as argument to the functions, in the form of a *continuation*.
- A continuation is an abstract representation of the control state of a program, most of the time in the form of a first-class function.
- Like SSA, CPS is often used in intermediate representation in compilers of functional languages.

CPS: examples

Direct style

```
(define (pyth x y)
 (sqrt (+ (* x x)(* y y))))
```

CPS style (k is the continuation)

```
(define (pyth& x y k)
 (*& x x (lambda (x2)
           (*& y y (lambda (y2)
                     (+& x2 y2 (lambda (x2py2)
                                 (sqrt& x2py2 k))))))))
(define (*& x y k)
 (k (* x y)))
(define (+& x y k)
 (k (+ x y)))
(define (sqrt& x k)
 (k (sqrt x)))
```

CPS

- The main interest of CPS is to make explicit several things that are typically implicit in functional languages: returns, intermediate values (= continuation arguments), order of argument evaluation...
- Like for SSA, the main interest is to ease optimizations.
- Theoretically, SSA and CPS are equivalent: a program in SSA form can be transformed into a CPS program and vice versa.
- Previous program can be rewritten as:

$$
\begin{aligned}
x2 &= x * x \\
y2 &= y * y \\
x2py2 &= x2 + y2 \\
res &= sqrt(x2py2)
\end{aligned}
$$

Outline

1. Intermediate representations

2. Illustration

3. Optimization

The intermediate language

We will illustrate the translation of typical high-level language constructions using the following low-level intermediate language:

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & [\,\mathit{Instructions}\,] \\
\\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
\\
\mathit{Instruction} & \rightarrow & \textbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \textbf{id}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Instruction} & \rightarrow & \texttt{LABEL}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \texttt{GOTO}\ \textbf{labelid} \\
\\
\mathit{Instruction} & \rightarrow & \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \mathit{Atom}\ \texttt{THEN}\ \textbf{labelid}\ \texttt{ELSE}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \texttt{CALL}\ \textbf{functionid}(\mathit{Args}) \\
\\
\mathit{Atom} & \rightarrow & \textbf{id} \\
\mathit{Atom} & \rightarrow & \textbf{num} \\
\mathit{Args} & \rightarrow & \textbf{id} \\
\mathit{Args} & \rightarrow & \textbf{id}\ ,\ \mathit{Args}
\end{array}
$$

Simplified three-address code, very close to machine code

See chapter 5 and 7 of (Mogensen, 2010) for full details

The intermediate language

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & [\ \mathit{Instructions}\ ] \\
 & & \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
 & & \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{id}\ \mathbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \mathbf{id} \\
 & & \\
\mathit{Atom} & \rightarrow & \mathbf{id} \\
\mathit{Atom} & \rightarrow & \mathbf{num}
\end{array}
$$

- All values are assumed to be integer
- Unary and binary operators include normal arithmetic and logical operations
- An atomic expression is either a variable or a constant
- $M[\mathit{Atom}] := \mathbf{id}$ is a transfer from a variable to memory
- $\mathbf{id} := M[\mathit{Atom}]$ is a transfer from memory to a variable

The intermediate language

$$
\begin{array}{lcl}
\textit{Instruction} & \rightarrow & \texttt{LABEL}\ \textbf{labelid} \\
\textit{Instruction} & \rightarrow & \texttt{GOTO}\ \textbf{labelid} \\
\\
\textit{Instruction} & \rightarrow & \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \textit{Atom}\ \texttt{THEN}\ \textbf{labelid}\ \texttt{ELSE}\ \textbf{labelid} \\
\textit{Instruction} & \rightarrow & \textbf{id} := \texttt{CALL}\ \textbf{functionid}(\textit{Args}) \\
\\
\textit{Atom} & \rightarrow & \textbf{id} \\
\textit{Atom} & \rightarrow & \textbf{num} \\
\\
\textit{Args} & \rightarrow & \textbf{id} \\
\textit{Args} & \rightarrow & \textbf{id}\ ,\ \textit{Args}
\end{array}
$$

- LABEL only marks a position in the program
- **relop** includes relational operators $\{=, \neq, <, >, \leq \text{ or } \geq\}$
- Arguments of a function call are variables and the result is assigned to a variable

Principle of translation

- Syntax-directed translation using several attributes:
  - Code returned as a synthesized attribute
  - Symbol tables passed as inherited attributes
  - Places to store intermediate values as synthesized or inherited attributes
- Implemented as recursive functions defined on syntax tree nodes (as for type checking)
- Since translation follows the syntax, it is done mostly independently of the context, which leads to suboptimal code
- Code is supposed to be optimized globally afterwards

Expressions

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \textbf{binop}\ Exp \\
Exp & \rightarrow & \textbf{id}(Exps) \\
 & & \\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

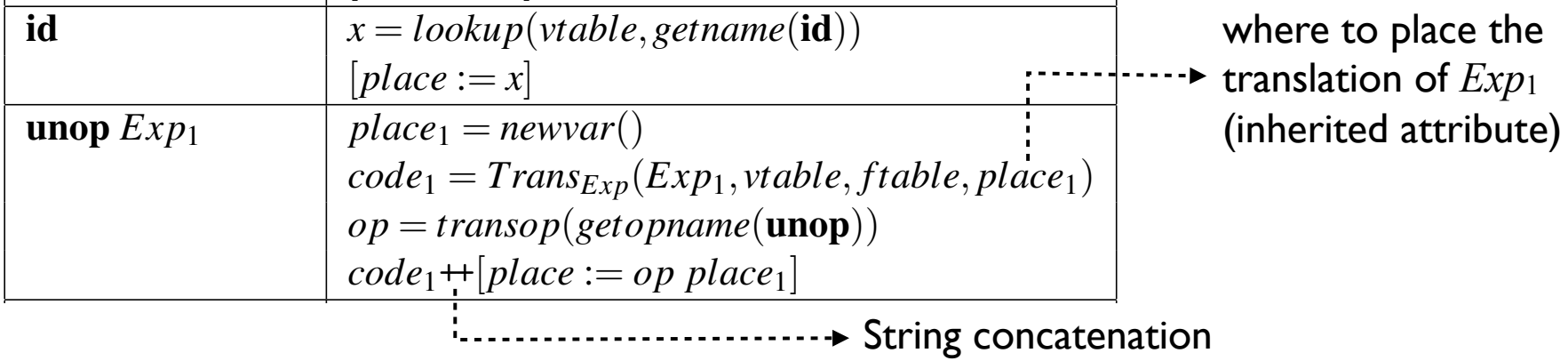
(source language grammar !)

Principle of translation:

- Every operation is stored in a new variable in the intermediate language, generated by a function *newvar*
- The new variables for sub-expressions are created by parent expression and passed to sub-expression as inherited attributes (synthesized attributes are also possible)

Expressions

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\textbf{unop}))$ $code_1 \mathbin{+\!\!+} [place := op\ place_1]$



- *getopname* retrieves the operator associated to the token **unop**. *transop* translates this operator into the equivalent operator in the intermediate language
- [*place* := *v*] is a string where *place* and *v* have been replaced by their values (in the compiler)
  - Example: if *place* = *t*14 and *v* = 42, [*place* := *v*] is the instruction [t14:=42].

Expressions: binary operators and function call

$Trans_{Exp}(Exp, vtable, ftable, place) = \texttt{case}\ Exp\ \texttt{of}$	
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\textbf{binop}))$ $code_1 ++ code_2 ++ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\textbf{id}))$ $code_1 ++ [place := \texttt{CALL}\ fname(a_1, \ldots, a_n)]$

Expressions: function arguments

$Trans_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
Exp , $Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 +\!\!+ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Expressions: example of translation

Translation of 3+f(x-y,z):

```
t1 := 3
                t4 := v0
                t5 := v1
           t3 := t4 - t5
           t6 := v2
      t2 := CALL _f(t3,t6)
t0 := t1+t2
```

Assuming that:

- x, y, and z are bound to variables v0, v1, and v2
- Expression is stored in t0
- New variables are generated as t1, t2, t3...
- Indentation indicates depth of call to $Trans_{Exp}$

Statements

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \mathit{Stat}\ ;\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \mathbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat}\ \texttt{else}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{while}\ \mathit{Cond}\ \texttt{do}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{repeat}\ \mathit{Stat}\ \texttt{until}\ \mathit{Cond}
\end{array}
$$

$$
\mathit{Cond} \rightarrow \mathit{Exp}\ \mathbf{relop}\ \mathit{Exp}
$$

Principle of translation:

- New unused labels are generated by the function *newlabel* (similar to *newvar*)
- These labels are created by parents and passed as inherited attributes

Statements: sequence of statements and assignment

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Stat_1$; $Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ code_2$
id $:= Exp$	$place = lookup(vtable, getname(\textbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$

Statements: conditions

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
`if` $Cond$ `then` $Stat_1$ `else` $Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ [\texttt{LABEL}\ label_1] ++ code_2$ $++ [\texttt{GOTO}\ label_3, \texttt{LABEL}\ label_2]$ $++ code_3 ++ [\texttt{LABEL}\ label_3]$

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1 ++ code_2 ++ [\texttt{IF}\ t_1\ op\ t_2\ \texttt{THEN}\ label_t\ \texttt{ELSE}\ label_f]$

Statements: while loop

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
`while` $Cond$ `do` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] ++ code_1$ $\quad ++ [\texttt{LABEL}\ label_2] ++ code_2$ $\quad ++ [\texttt{GOTO}\ label_1, \texttt{LABEL}\ label_3]$

Logical operators

- Logical conjunction, disjunction, and negation are often available to define conditions
- Two ways to implement them:
  - Usual arithmetic operators: arguments are evaluated and then the operators is applied. Example in C: bitwise operators: '&' and '|'.
  - Sequential logical operators: the second operand is not evaluated if the first determines the result (lazy or short-circuit evaluation). Example in C: logical operators '&&' and '||'.
- First type is simple to implement:
  - by allowing any expression as condition

$$Cond \rightarrow Exp$$

  - by including '&', '|', and '!' among binary and unary operators
- Second one requires more modifications

Sequential logical operators

$Cond \rightarrow Exp$ **relop** Exp
$Cond \rightarrow$ `true`
$Cond \rightarrow$ `false`
$Cond \rightarrow$ `!` $Cond$
$Cond \rightarrow Cond$ `&&` $Cond$
$Cond \rightarrow Cond$ `||` $Cond$

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
`true`	$[$`GOTO` $label_t]$
`false`	$[$`GOTO` $label_f]$
`!` $Cond_1$	$Trans_{Cond}(Cond_1, label_f, label_t, vtable, ftable)$
$Cond_1$ `&&` $Cond_2$	$arg_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, arg_2, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1 ++ [$`LABEL` $arg_2] ++ code_2$
$Cond_1$ `||` $Cond_2$	$arg_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, arg_2, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1 ++ [$`LABEL` $arg_2] ++ code_2$

Other statements

More advanced control statements:

- Goto and labels: labels are stored in the symbol table (and associated with intermediate language labels). Generated as soon as a jump or a declaration is met (to avoid one additional pass)
- Break/exit: pass an additional (inherited) attribute to the translation function of loops with the label a break/exit should jump to. A new label is passed when entering a new loop.
- Case/switch-statements: translated with nested if-then-else statements.
- ...

Arrays

Language can be extended with one-dimensional arrays:

$$\begin{array}{lcl} Exp & \rightarrow & Index \\ Stat & \rightarrow & Index := Exp \\ Index & \rightarrow & \mathbf{id}[Exp] \end{array}$$

Principle of translation:

- Arrays can be allocated statically (at compile-time) or dynamically (at run-time)
- Base address of the array is stored as a constant in the case of static allocation, or in a variable in the case of dynamic allocation
- The symbol table binds the array name to the constant or variable containing its address

Arrays: translation

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
$Index$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $code_1 ++ [place := M[address]]$

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Index := Exp$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 ++ code_2 ++ [M[address] := t]$

$Trans_{Index}(Index, vtable, ftable) =$ `case` $Index$ `of`	
$\mathbf{id}[Exp]$	$base = lookup(vtable, getname(\mathbf{id}))$ $t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_2 = code_1 ++ [t := t * 4, t := t + base]$ $(code_2, t)$

(Assuming arrays are indexed starting at 0 and integers are 32 bits long)

Multi-dimensional arrays

$$\begin{aligned} Index &\rightarrow \mathbf{id}[Exp] \\ Index &\rightarrow Index[Exp] \end{aligned}$$

Principle of translation:

- Two ways to represent a 2-dimensional array in linear memory:
  - Row-major order: one row at a time. For a 3×2 array: a[0][0], a[0][1], a[1][0], a[1][1], a[2][0], a[2][1]
  - Column-major order: one column at a time. For a 3×2 array: a[0][0], a[1][0], a[2][0], a[0][1], a[1][1], a[2][1]
- Generalization: if $dim_0, dim_1, \ldots, dim_{n-1}$ are the sizes of the dimensions in a n-dimensional array, the element $[i_0][i_1]\ldots[i_{n-1}]$ has the address:
  - Row-major:
    $base + ((\ldots(i_0 \cdot dim_1 + i_1) \cdot dim_2 \ldots + i_{n-2}) \cdot dim_{n-1} + i_{n-1}) \cdot size$
  - Column-major:
    $base + ((\ldots(i_{n-1} \cdot dim_0 + i_{n-2}) \cdot dim_1 \ldots + i_1) \cdot dim_{n-2} + i_0) \cdot size$
- Dimension sizes are stored as constant (static), in variables or in memory next to the array data (dynamic)

Multi-dimensional arrays: translation

$Trans_{Index}(Index, vtable, ftable) =$
$(code_1, t, base, []) = Calc_{Index}(Index, vtable, ftable)$ $code_2 = code_1 ++ [t := t * 4, t := t + base]$ $(code_2, t)$

$Calc_{Index}(Index, vtable, ftable) =$ `case` $Index$ `of`	
$\mathbf{id}[Exp]$	$(base, dims) = lookup(vtable, getname(\mathbf{id}))$ $t = newvar()$ $code = Trans_{Exp}(Exp, vtable, ftable, t)$ $(code, t, base, tail(dims))$
$Index[Exp]$	$(code_1, t_1, base, dims) = Calc_{Index}(Index, vtable, ftable)$ $dim_1 = head(dims)$ $t_2 = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t_2)$ $code_3 = code_1 ++ code_2 ++ [t_1 := t_1 * dim_1, t_1 := t_1 + t_2]$ $(code_3, t_1, base, tail(dims))$

(Assume dimension sizes are stored in the symbol table, as constant or variable)

Other structures

- Floating point values: can be treated the same way as integers (assuming the intermediate language has specific variables and operators for floating point numbers)
- Records/structures: allocated in a similar way as arrays
  - Each field is accessed by adding an offset to the base-address of the record
  - Base-addresses and offsets for each field are stored in the symbol table for all record-variables
- Strings: similar to arrays of bytes but with a length that can vary at run-time
- ...

Variable declaration

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \mathit{Decl}\ ;\ \mathit{Stat} \\
\mathit{Decl} & \rightarrow & \texttt{int}\ \mathbf{id} \\
\mathit{Decl} & \rightarrow & \texttt{int}\ \mathbf{id}[\mathbf{num}]
\end{array}
$$

Principle of translation:

- Information about where to find scalar variables (e.g. integer) and arrays after declaration is stored in the symbol table
- Allocations can be done in many ways and places (static, dynamic, local, global...)

Variable declaration

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Decl$; $Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1 ++ code_2$

$Trans_{Decl}(Decl, vtable) =$ `case` $Decl$ `of`	
`int` **id**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([], vtable_1)$
`int` **id**[**num**]	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([t_1 := HP, HP := HP + (4 * getvalue(\mathbf{num}))], vtable_1)$

(Assumes scalar variables are stored in intermediate language variables and arrays are dynamically allocated on the heap, with their base-addresses stored in a variable. *HP* points to the first free position of the heap.)

Comments

- Needs to add error checking in previous illustration (array index out of bounds in arrays, wrong number of dimensions, memory/heap overflow, etc.)
- In practice, results of translation are not returned as strings but either:
  - output directly into an array or a file
  - or stored into a structure (translation tree or linked list)

  The latter allows subsequent code restructuring during optimization
- We have not talked about:
  - memory organization: typically subdivided into static data (for static allocation), heap (for dynamic allocation) and stack (for function calls)
  - translation of function calls: function arguments, local variables, and return address are stored on the stack (similar to what you have seen in INFO-0012, computation structures)

Outline

1. Intermediate representations

2. Illustration

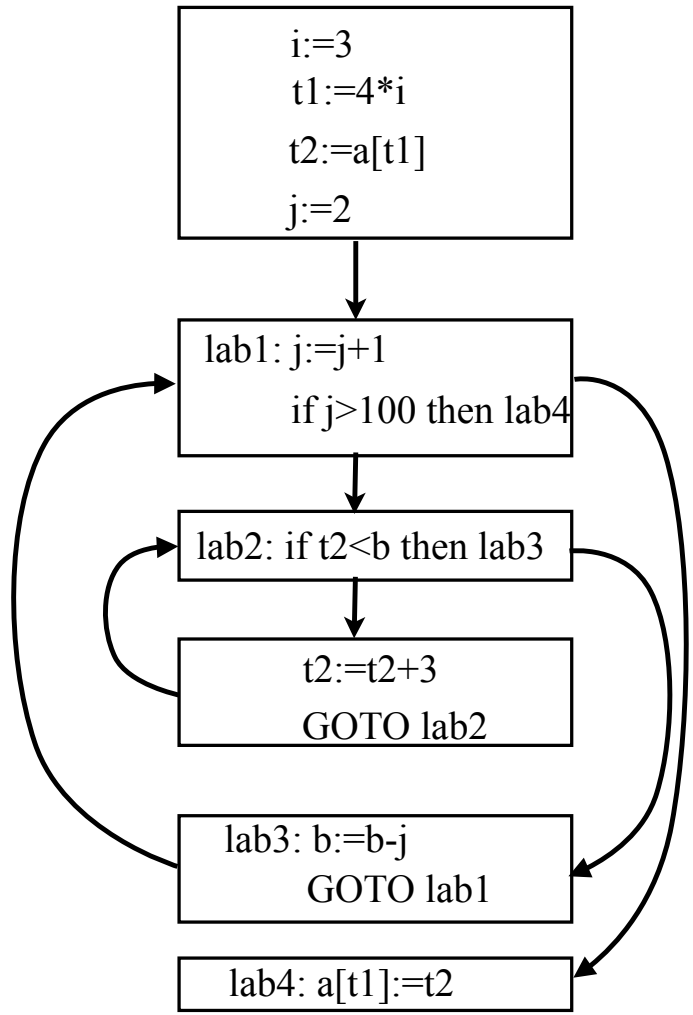
3. Optimization

IR code optimization

- IR code generation is usually followed by code optimization
- Why?
  - IR generation introduces redundancy
  - To compensate for laziness of programmers
- Improvement rather than optimization since optimization is undecidable
- Challenges in optimization:
  - Correctness: should not change the semantic of the program
  - Efficiency: should produce IR code as efficient as possible
  - Computing times: should not take too much time to optimize
- What to optimize?
  - Computing times
  - Memory usage
  - Power consumption
  - ...

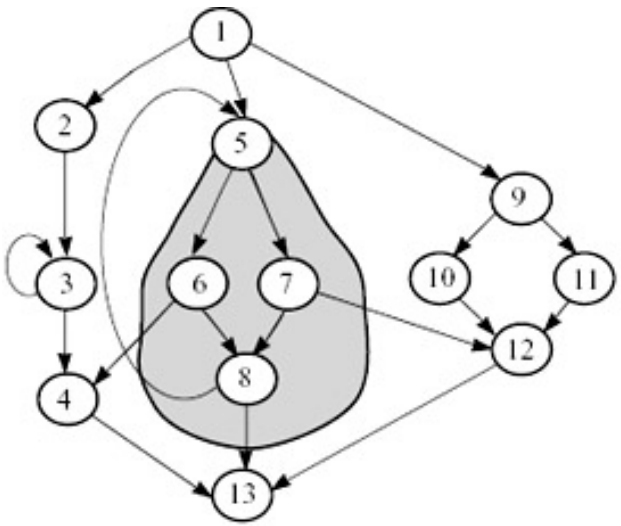
Control-flow graph

- A basic block is a series of IR instructions where:
  - there is one entry point into the basic block, and
  - there is one exit point out of the basic block.
- Control-flow graph: nodes are basic blocks and edges are jumps between blocks



Control-flow graph and SSA

- The control-flow graph (CFG) can be used to determine where to introduce Φ functions when deriving a SSA form:
  - A node A (basic block) of the CFG *strictly dominates* a node B if it is impossible to reach B without going through A. A *dominates* B if A stricly dominates B or $A = B$.
  - B is in the *dominance frontier* of A if A does not strictly dominate B, but dominates some immediate predecessor of B.
  - Whenever node A contains a definition of a variable x, any node B in the dominance frontier of A needs a Φ function for x.
- There exist an efficient algorithm to find the dominance frontier of a node



4,5,12,13 are in the dominance frontier of 5

(Appel)

Local optimizations

Local optimization: optimization within a single basic block

Examples:

- Constant folding: evaluation at compile-time of expressions whose operands are constant
  - 10+2*3 $\rightarrow$ 16
  - [If 1 then Lab1 Else Lab2] $\rightarrow$ [GOTO Lab1]

- Constant propagation: if a variable is assigned a constant, then propagate the constant into each use of the variable
  - [x:=4;t:=y*x;] can be transformed into [t:=y*4;] if x is not used later

Local optimizations

Examples:

- Copy propagation: similar to constant propagation but generalized to non constant values

```
tmp2 = tmp1;
tmp3 = tmp2 * tmp1;
tmp4 = tmp3;
tmp5 = tmp3 * tmp2;
c = tmp5 + tmp4;
```

```
tmp3 = tmp1 * tmp1;
tmp5 = tmp3 * tmp1;
c = tmp5 + tmp3;
```

- Dead code elimination: remove instructions whose result is never used
  - Example: Remove [tmp1=tmp2+tmp3;] if tmp1 is never used

Local optimizations

Examples:

- Common subexpression elimination: if two operations produce the same results, compute the result once and reference it the second time
  - Example: in `a[i]=a[i]+2`, the address of `a[i]` is computed twice. When translating, do it once and store the result in a temporary variable

- Code moving/hoisting: move outside of a loop all computations independent of the variables that are changing inside the loop
  - Example: part of the computation of the address for `a[i][j]` can be removed from this loop

```
while (j<k) {
  sum = sum + a[i][j];
  j++;
}
```

IR code optimization

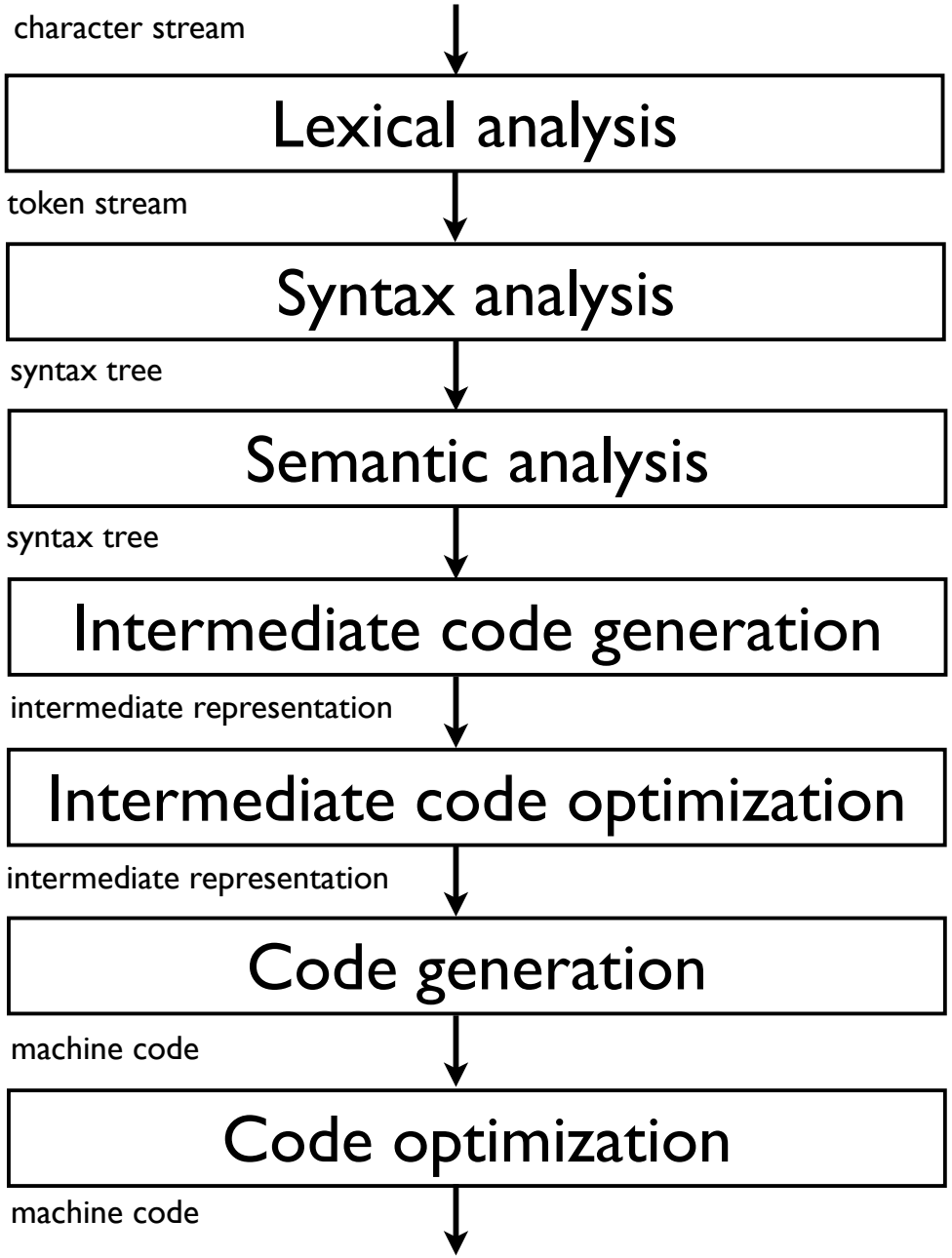
- Local optimizations can be interleaved in different ways and applied several times each
- Optimal optimization order is very difficult to determine

- Global optimization: optimization across basic blocks
  - Implies performing data-flow analysis, i.e., determine how values propagate through the control-flow graph
  - More complicated than local optimization

Part 6

Code generation

Structure of a compiler



Final code generation

- At this point, we have optimized intermediate code, from which we would like to generate the final code
- By final code, we typically mean assembly language of the target machine
- Goal of this stage:
  - Choose the appropriate machine instructions to translate each intermediate representation instruction
  - Handle finite machine resources (registers, memory, etc.)
  - Implement low-level details of the run-time environment
  - Implement machine-specific code optimization
- This step is very machine-specific
- In this course, we will only mention some typical and general problems

Short tour on machine code

- RISC (Reduced Instruction Set Computer)
  - E.g.: PowerPC, Sparc, MIPS (embedded systems), ARM...
  - Many registers, 3-address instructions, relatively simple instruction sets
- CISC (Complex Instruction Set Computer)
  - E.g.: x86, x86-64, amd64...
  - Few registers, 2-address instructions, complex instruction sets
- Stack-based computer:
  - E.g.: Not really used anymore but Java's virtual machine is stack-based
  - No register, zero address instructions (operands on the stack)
- Accumulator-based computer:
  - E.g.: First IBM computers were accumulator-based
  - One special register (the accumulator), one address instructions, other registers used in loops and address specification

Outline

1. Introduction
2. Instruction selection
3. Register allocation
4. Memory management

Instruction selection

- One needs to map one or several instructions of the intermediate representation into one or several instructions of the machine language
- Complexity of the task depends on:
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code
- Examples of problems:
  - Conditional jumps
  - Constants
  - Complex instructions

Example: Conditional jumps

- Conditional jumps in our intermediate language are of the form:
  IF id relop *Atom* THEN labelid ELSE labelid

- Conditional jumps might be different on some machines:
  - One-way branch instead of two-way branches

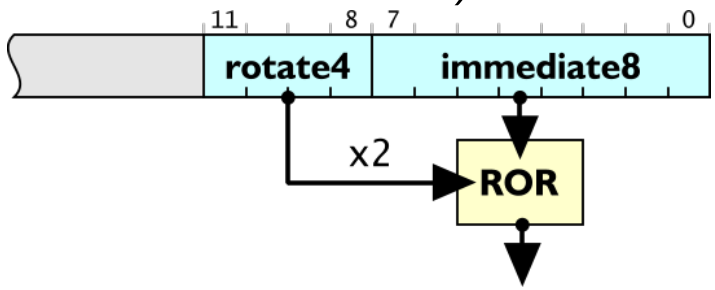
    IF c THEN l_t ELSE l_f

    ```
    branch_if_c   l_r
    jump          l_f
    ```

  - Condition such as "id relop *Atom*" might not be allowed. Then, compute the condition and store it in a register
  - There might exist special registers for conditions
  - ...

Example: Constants

- IR allows arbitrary constants as operands to binary or unary operators
- This is not always the case in machine code
  - MIPS allows only 16-bit constants in operands (even though integers are 32 bits)
  - On the ARM, a constant can only be a 8-bit number positioned at any even bit boundary (within a 32-bit word)



http://www.davespace.co.uk/arm/

- If a constant is too big, translation requires to build the constant into some register
- If the constant is used within a loop, its computation should be moved outside

Exploiting complex instructions

- If we do not care about efficiency, instruction selection is straightforward:
  - Write a code skeleton for every IR instruction
  - Example in MIPS assembly:

    $$t_2 := t_1 + 116 \quad \Rightarrow \quad \text{addi r2,r1,116}$$

    (where r2 and r1 are the registers chosen for t_2 and t_1)

- Most processors (even RISC-based) have complex instructions that can translate several IR instructions at once
  - Examples in MIPS assembly:

    $$t_2 := t_1 + 116 \quad \Rightarrow \quad \text{lw r3, 116(r1)}$$
    $$t_3 := M[t_2]$$

    (where r3 and r1 are the registers chosen for t_3 and t_1 resp. and assuming that t_2 will not be used later)
- For efficiency reason, one should exploit them

Code generation principle

- Determine for each variable whether it is dead after a particular use (liveness analysis, see later)

$$t_2 := t_1 + 116$$
$$t_3 := M[t_2^{last}]$$

- Associate an address (register, memory location...) to each variable (register allocation, see later)
- Define an instruction set description, i.e., a list of pairs of:
  - pattern: a sequence of IR instructions

$$t := r_s + k$$
$$r_t := M[t^{last}]$$

  - replacement: a sequence of machine-code instruction translating the pattern

$$\text{lw } r_t, k(r_s)$$

- Use pattern matching to do the translation

Illustration

Pattern/replacement pairs for a subset of the MIPS instruction set

$t := r_s + k$, $r_t := M[t^{last}]$	`lw`	r_t, $k(r_s)$
$r_t := M[r_s]$	`lw`	r_t, $0(r_s)$
$r_t := M[k]$	`lw`	r_t, k(R0)
$t := r_s + k$, $M[t^{last}] := r_t$	`sw`	r_t, $k(r_s)$
$M[r_s] := r_t$	`sw`	r_t, $0(r_s)$
$M[k] := r_t$	`sw`	r_t, k(R0)
$r_d := r_s + r_t$	`add`	r_d, r_s, r_t
$r_d := r_t$	`add`	r_d, R0, r_t
$r_d := r_s + k$	`addi`	r_d, r_s, k
$r_d := k$	`addi`	r_d, R0, k

MIPS instructions:

- lw r,k(s): $r = M[s + k]$
- sw r,k(s): $M[s + k] = r$
- add r,s,t: $r = s + t$
- addi r,s,k: $r = s + k$ where k is a constant
- R0: a register containing the constant 0

(Mogensen)

Illustration

IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$		beq	$r_s, r_t, label_t$
	$label_f$:		
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_t$		bne	$r_s, r_t, label_f$
	$label_t$:		
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$		beq	$r_s, r_t, label_t$
		j	$label_f$
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$		slt	r_d, r_s, r_t
		bne	r_d, R0, $label_t$
	$label_f$:		
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_t$		slt	r_d, r_s, r_t
		beq	r_d, R0, $label_f$
	$label_t$:		
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$		slt	r_d, r_s, r_t
		bne	r_d, R0, $label_t$
		j	$label_f$
LABEL $label$	$label$:		

MIPS instructions:

- beq r,s,lab: branch to lab if r=s
- bne r,s,lab: branch to lab if r≠s
- slt r,s,t: $r = (s < t)$
- j l: unconditional jump

(Mogensen)

Pattern matching

- A pattern should be defined for every single IR instruction (otherwise it would not be possible to translate some IR code)
- A *last* in a pattern can only be matched by a *last* in the IR code
- But any variable in a pattern can match a *last* in the IR code
- If patterns overlap, there are potentially several translations for the same IR code
- One wants to find the best possible translation (e.g., the shortest or the fastest)
- Two approaches:
  - **Greedy:** order the pairs so that longer patterns are listed before shorter ones and at each step, use the first pattern that matches a prefix of the IR code
  - **Optimal:** associate a cost to each replacement and find the translation that minimizes the total translation cost, e.g. using dynamic programming

Illustration

Using the greedy approach:

IR code		MIPS code
$a := a + b^{last}$		add a, a, b
$d := c + 8$		sw $a, 8(c)$
$M[d^{last}] := a$	$\Rightarrow$	
IF $a = c$ THEN $label_1$ ELSE $label_2$		beq $a, c, label_1$
LABEL $label_2$	$label_2$:	

Outline

1. Introduction

2. Instruction selection

3. Register allocation

4. Memory management

Register allocation

- In the IR, we assumed an unlimited number of registers (to ease IR code generation)
- This is obviously not the case on a physical machine (typically, 5-10 general-purpose registers for a CISC architecture, $>$15 for a RISC architecture)

- Registers can be accessed quickly and operations can be performed on them directly
- Using registers intelligently is therefore a critical step in any compiler (can make a difference in orders of magnitude)

- Register allocation is the process of assigning variables to registers and managing data transfer in and out of the registers

Challenges in register allocation

- Registers are scarce
  - Often substantially more IR variables than registers
  - Need to find a way to reuse registers whenever possible
- Register management is sometimes complicated
  - Each register is made of several small registers (x86)
  - There are specific registers which need to be used for some instructions (x86)
  - Some registers are reserved for the assembler or operating systems (MIPS)
  - Some registers must be reserved to handle function calls (all)
- Here, we assume only some number of indivisible, general-purpose registers (MIPS-style)

A direct solution

- Idea: store every value in main memory, loading values only when they are needed.
- To generate a code that performs some computation:
  - Generate load instructions to retrieve the values from main memory into registers
  - Generate code to perform the computation on the registers
  - Generate store instructions to store the result back into main memory
- Example: (with a,b,c,d stored resp. at fp-8, fp-12, fp-16, fp-20)

$$
\begin{array}{lcl}
a := b + c & & \text{lw } t_0, -12(fp) \\
d := a & \Rightarrow & \text{lw } t_1, -16(fp) \\
c := a + d & & \text{add } t_2, t_0, t_1 \\
 & & \text{sw } t_2, -8(fp) \\
\hline
 & & \text{lw } t_0, -8(fp) \\
 & & \text{sw } t_0, -20(fp) \\
\hline
 & & \text{lw } t_0, -8(fp) \\
 & & \text{lw } t_1, -20(fp) \\
 & & \text{add } t_2, t_0, t_1 \\
 & & \text{sw } t_2, -16(fp)
\end{array}
$$

A direct solution

- Advantage: very simple, translation is straighforward, never runout of registers
- Disadvantage: very inefficient, waste space and time

- Better allocator should:
  - try to reduce memory load/store
  - reduce total memory usage
- Need to answer two questions:
  - Which register do we put variables in?
  - What do we do when we run out of registers?

Liveness analysis

- A variable is live at some point in the program if its value may be read later before it is written. It is dead if there is no way its value can be used in the future.
- Two variables can share a register if there is no point in the program where they are both live
- Liveness analysis is the process of determining the live or dead statuses of all variables throughout the (IR) program

- Informally: For an instruction I and a variable t
  - If t is used in I, then t is live at the start of I
  - If t is assigned a value in I (and does not appear in the RHS of I), then t is dead at the start of the I
  - If t is live at the end of I and I does not assign a value to t, then t is live at the start of I
  - t is live at the end of I if it is live at the start of any of the immediately succeeding instructions

Liveness analysis: control-flow graph

First step: construct the control-flow graph

- For each instruction numbered i, one defines $succ[i]$ as follows:
  - If instruction j is just after i and i is neither a GOTO or IF-THEN-ELSE instruction, then j is in $succ[i]$
  - If i is of the form GOTO l, the instruction with label l is in $succ[i]$.
  - If i is IF p THEN l_t ELSE l_f, instructions with label l_t and l_f are both in $succ[i]$

- The third rule loosely assumes that both outcomes of the IF-THEN-ELSE are possible, meaning that some variables will be claimed live while they are dead (not really a problem)

Liveness analysis: control-flow graph

Example (Computation of Fibonacci(n) in a)

```
1:  a := 0
2:  b := 1
3:  z := 0
4:  LABEL loop
5:  IF n = z THEN end ELSE body
6:  LABEL body
7:  t := a + b
8:  a := b
9:  b := t
10: n := n − 1
11: z := 0
12: GOTO loop
13: LABEL end
```

i	$succ[i]$
1	2
2	3
3	4
4	5
5	6, 13
6	7
7	8
8	9
9	10
10	11
11	12
12	4
13	

Liveness analysis: *gen* and *kill*

For each IR instruction, we define two functions:

- $gen[i]$: set of variables that may be read by instruction i
- $kill[i]$: set of variables that may be assigned a value by instruction i

Instruction i	$gen[i]$	$kill[i]$
`LABEL` l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y,z\}$	$\{x\}$
$x := y$ **binop** k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	$\emptyset$	$\{x\}$
$M[x] :=$ y	$\{x,y\}$	$\emptyset$
$M[k] :=$ y	$\{y\}$	$\emptyset$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` x **relop** y `THEN` l_t `ELSE` l_f	$\{x,y\}$	$\emptyset$
$x :=$ `CALL` $f(args)$	$args$	$\{x\}$

Liveness analysis: *in* and *out*

- For each program instruction i, we use two sets to hold liveness information:
  - $in[i]$: the variables that are live before instruction i
  - $out[i]$: the variables that are live at the end of i
- *in* and *out* are defined by these two equations:

$$\begin{aligned} in[i] &= gen[i] \cup (out[i] \setminus kill[i]) \\ out[i] &= \bigcup_{j \in succ[i]} in[j] \end{aligned}$$

- These equations can be solved by fixed-point iterations:
  - Initialize $in[i]$ and $out[i]$ to empty sets
  - Iterate over instructions (in reverse order, evaluating *out* first) until convergence (i.e., no change)
- For the last instruction ($succ[i] = \emptyset$), $out[i]$ is a set of variables that are live at the end of the program (i.e., used subsequently)

Illustration

```
1:  a := 0
2:  b := 1
3:  z := 0
4:  LABEL loop
5:  IF n = z THEN end ELSE body
6:  LABEL body
7:  t := a + b
8:  a := b
9:  b := t
10: n := n - 1
11: z := 0
12: GOTO loop
13: LABEL end
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13			

(Mogensen)

(We can assume that $out[13] = \{a\}$)

Illustration

	Initial		Iteration 1		Iteration 2		Iteration 3	
i	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$
1			n,a	n	n,a	n	n,a	n
2			n,a,b	n,a	n,a,b	n,a	n,a,b	n,a
3			n,z,a,b	n,a,b	n,z,a,b	n,a,b	n,z,a,b	n,a,b
4			n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b
5			a,b,n	n,z,a,b	a,b,n	n,z,a,b	a,b,n	n,z,a,b
6			a,b,n	a,b,n	a,b,n	a,b,n	a,b,n	a,b,n
7			b,t,n	a,b,n	b,t,n	a,b,n	b,t,n	a,b,n
8			t,n	b,t,n	t,n,a	b,t,n	t,n,a	b,t,n
9			n	t,n	n,a,b	t,n,a	n,a,b	t,n,a
10				n	n,a,b	n,a,b	n,a,b	n,a,b
11					n,z,a,b	n,a,b	n,z,a,b	n,a,b
12					n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b
13			a	a	a	a	a	a

(Mogensen)

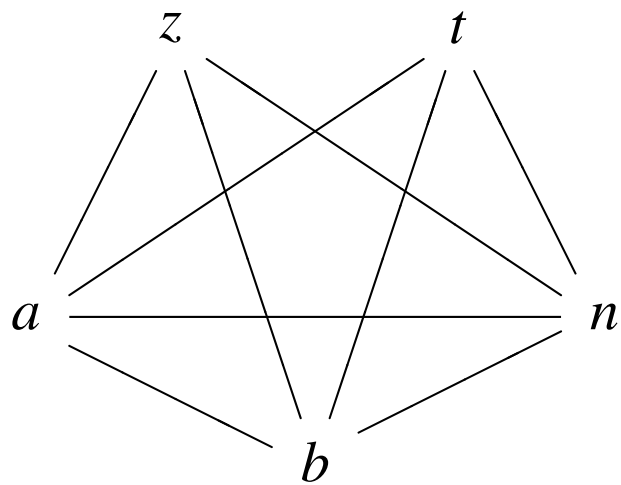
(From instruction 13 to instruction 1)

Interference

- A variable x interferes with another variable y if there is an instruction i such that $x \in \mathit{kill}[i]$, $y \in \mathit{out}[i]$ and instruction i is not $x := y$
- Two variables can share a register precisely if neither interferes with the other.
- Note: This is different from $x \in \mathit{out}[i]$ and $y \in \mathit{out}[i]$ (ie., x and y live simultaneously)
  - if x is in $\mathit{kill}[i]$ and not in $\mathit{out}[i]$ (because x is never used after an assignment), then it should interfere with $y \in \mathit{out}[i]$, otherwise if x and y share the same register, an assignment to x will overwrite the live variable y.
- Interference graph: undirected graph where each node is a variable and two variables are connected if they interfere

Illustration

Instruction	Left-hand side	Interferes with
1	a	n
2	b	n, a
3	z	n, a, b
7	t	b, n
8	a	t, n
9	b	n, a
10	n	a, b
11	z	n, a, b



(Mogensen)

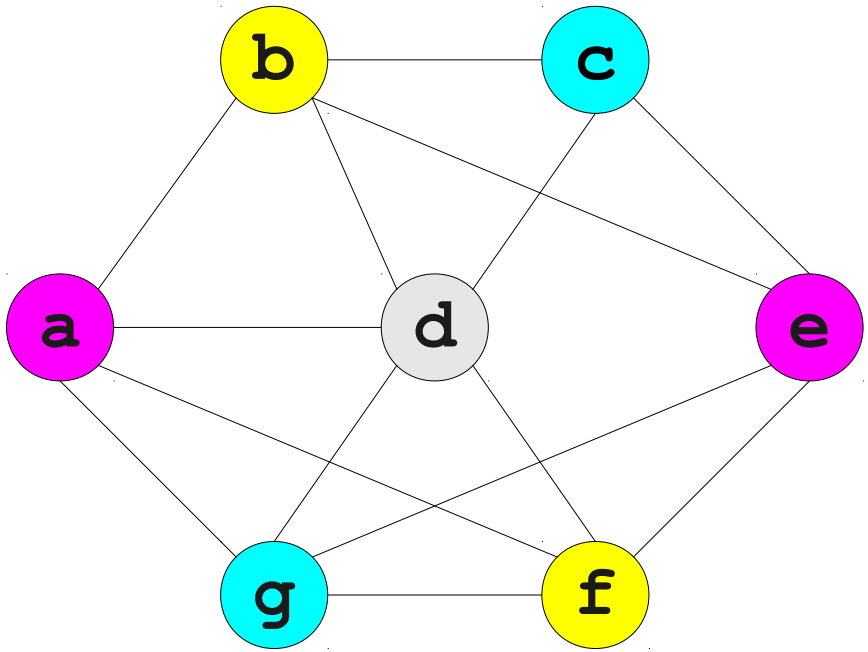
Register allocation

- Global register allocation: we assign to a variable the same register throughout the program (or procedure)
- How to do it? Assign a register number (among N) to each node of the interference graph such that
  - Two nodes that are connected have different register numbers
  - The total number of different registers is no higher than the number of available registers
- This is a problem of graph coloring (where color number = register number), which is known to be NP-complete
- Several heuristics have been proposed

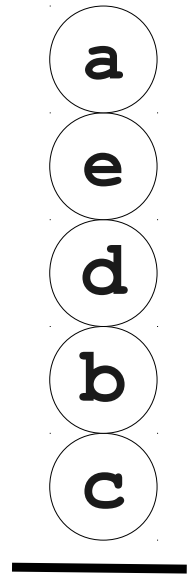
Chaitin's algorithm

- A heuristic linear algorithm for k-coloring a graph
- Algorithm:
  - Select a node with fewer than k outgoing edges
  - Remove it from the graph
  - Recursively color the rest of the graph
  - Add the node back in
  - Assign it a valid color
- Last step is always possible since the removed node has less than k neighbors in the graph
- Implementation: nodes are pushed on a stack as soon as they are selected

Illustration



Stack of nodes

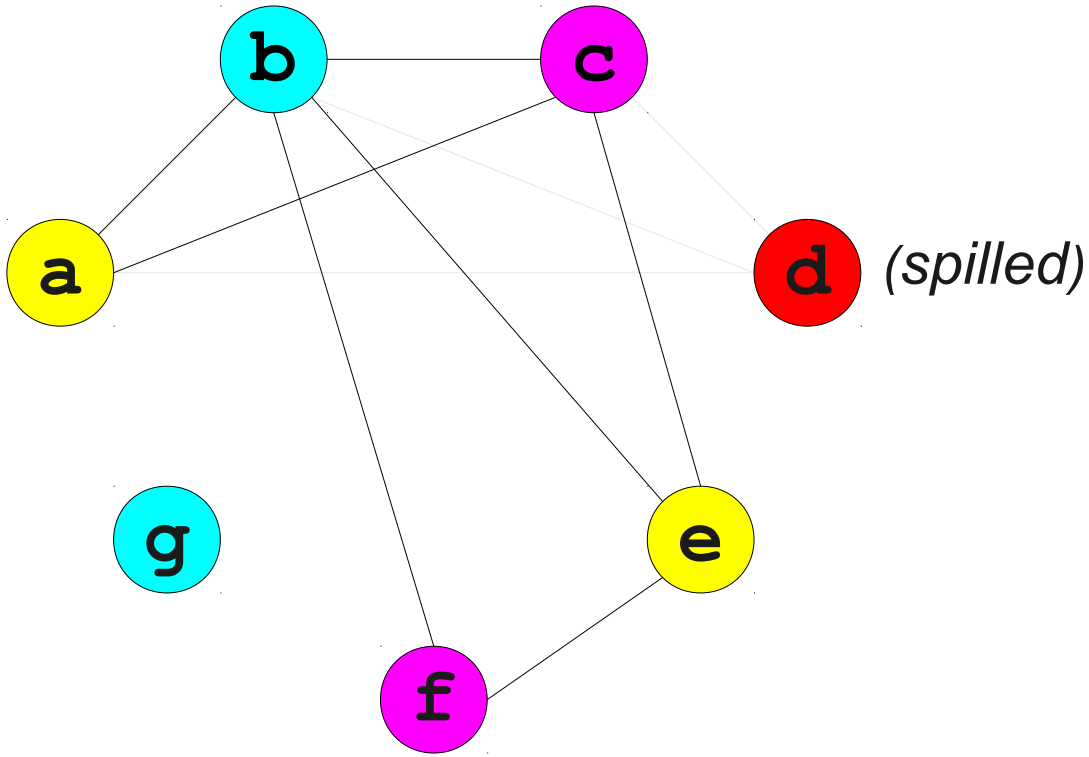


(Keith Schwarz)

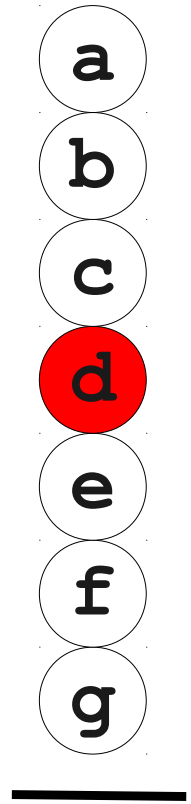
Chaitin's algorithm

- What if we can not find a node with less than k neighbors?
- Choose and remove an arbitrary node, marking it as "troublesome"
- When adding node back in, it may still be possible to find a valid color
- Otherwise, we will have to store it in memory.
  - This is called spilling.

Illustration



Stack of nodes



Registers

(Keith Schwarz)

Spilling

- A spilled variable is stored in memory
- When we need a register for a spilled variable v, temporarily evict a register to memory (since registers are supposed to be exhausted)
- When done with that register, write its value to the storage spot for v (if necessary) and load the old value back

- Heuristics to choose the variable/node to spill:
  - Pick one with close to N neighbors (increasing the chance to color it)
  - Choose a node with many neighbors with close to N neighbors (increase the chance of less spilling afterwards)
  - Choose a variable that's not costly to spill (by looking at the program)

Register allocation

- We only scratched the surface of register allocation
- Many heuristics exist as well as different approaches (not using graph coloring)
- GCC uses a variant of Chaitin's algorithm

Outline

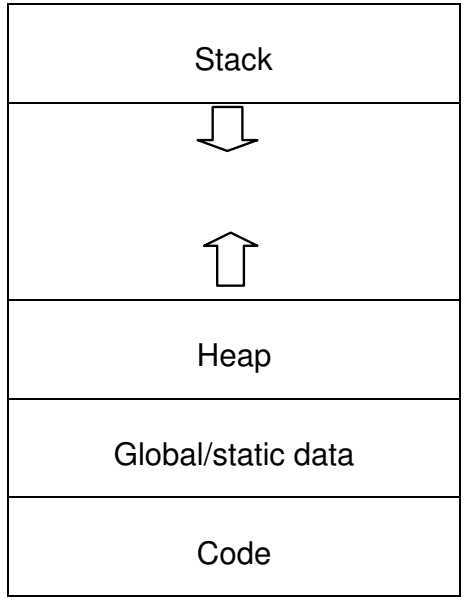
1. Introduction

2. Instruction selection

3. Register allocation

4. Memory management

Memory organization



Memory is generally divided into four main parts:

- Code: contains the code of the program
- Static data: contains static data allocated at compile-time
- Stack: used for function calls and local variables
- Heap: for the rest (e.g., data allocated at run-time)

Computers have registers that contain addresses that delimit these different parts

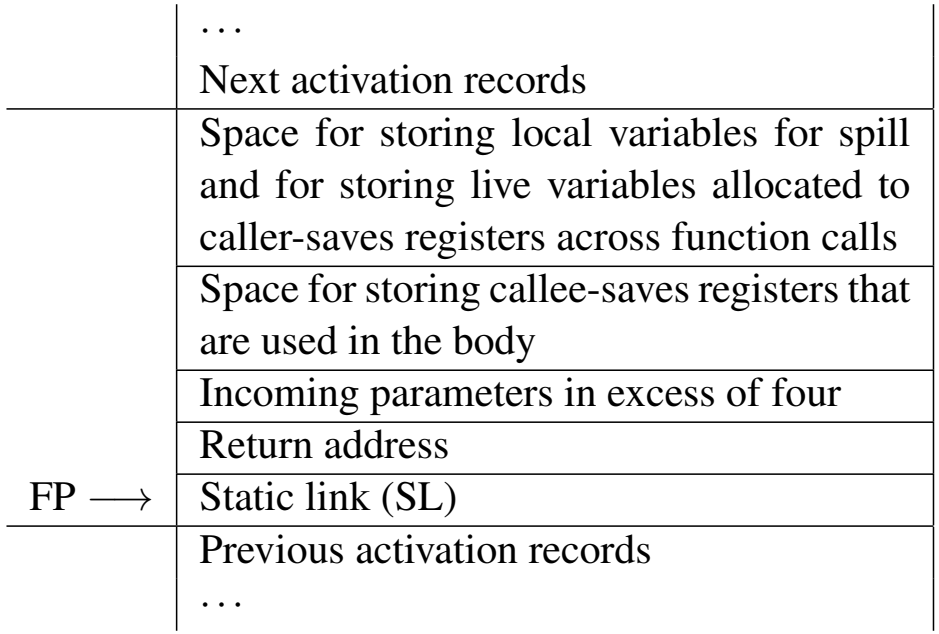
Static data

- Contains data allocated at compile-time
- Address of such data is then hardwired in the generated code
- Used e.g. in C to allocate global variables
- There are facilities in assemblers to allocate such space:
  - Example to allocate an array of 4000 bytes

```
        .data           # go to data area for allocation
baseofA:                # label for array A
        .space 4000     # move current-address pointer up 4000 bytes
        .text           # go back to text area for code generation
```

- Limitations:
  - size of the data must be known at compile-time
  - Never freed even if the data is only used a fraction of time

Stack



- Mainly used to store activation records for function calls
- But can be used to allocate arrays and other data structures (e.g., in C, to allocate *local* arrays)

- Allocation is quick and easy
- But sizes of arrays need to be known at compile-time and can only be used for local variables (space is freed when the function returns)

Heap

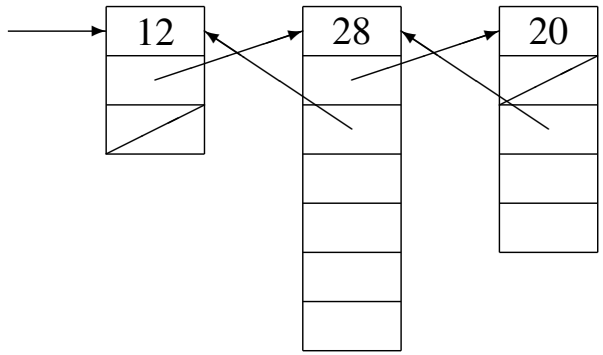
- Used for dynamic memory allocations
- Size of arrays or structures need not to be known at compile-time
- Array sizes can be increased dynamically
- Two ways to manage data allocation/deallocation:
  - Manual memory management
  - Automatic memory management (or garbage collection)

Manual memory management

- The user is responsible for both data allocation and deallocation
  - In C: malloc and free
  - In object oriented languages: object constructors and destructors
- Advantages:
  - Easier to implement than garbage collection
  - The programmer can exercise precise control over memory usage (allows better performances)
- Limitations
  - The programmer *has to* exercise precise control over memory usage (tedious)
  - Easily leads to troublesome bugs: memory leaks, double frees, use-after-frees...

A simple implementation

- Space is allocated by the operating system and then managed by the program (through library functions such as malloc and free in C)
- A free list is maintained with all current free memory blocks (initially, one big block)



- Malloc:
  - Search through the free list for a block of sufficient size
  - If found, it is possibly split in two with one removed from free list
  - If not found, ask operating system for a new chunk of memory
- Free:
  - Insert the block back into the free list
- Allocation is linear in the size of the free list, deallocation is done in constant time

A simple implementation

- Block splitting leads to memory fragmentation
  - The free list will eventually accumulate many small blocks
  - Can be solved by joining consecutive freed blocks
  - Makes free linear in free list size
- Complexity of malloc can be reduced
  - Limit block sizes to power of 2 and have a free list for each size
  - Look for a block of the power of 2 just greater than searched size.
  - If not available, take the next bigger block available and split it in two repetitively until size is correct.
  - Makes malloc logarithmic in heap size in the worst case.
- Array resizing can be allowed by using indirection nodes
  - When array is resized, it is copied into a new (bigger) block
  - Indirection node address is updated accordingly

Garbage collection

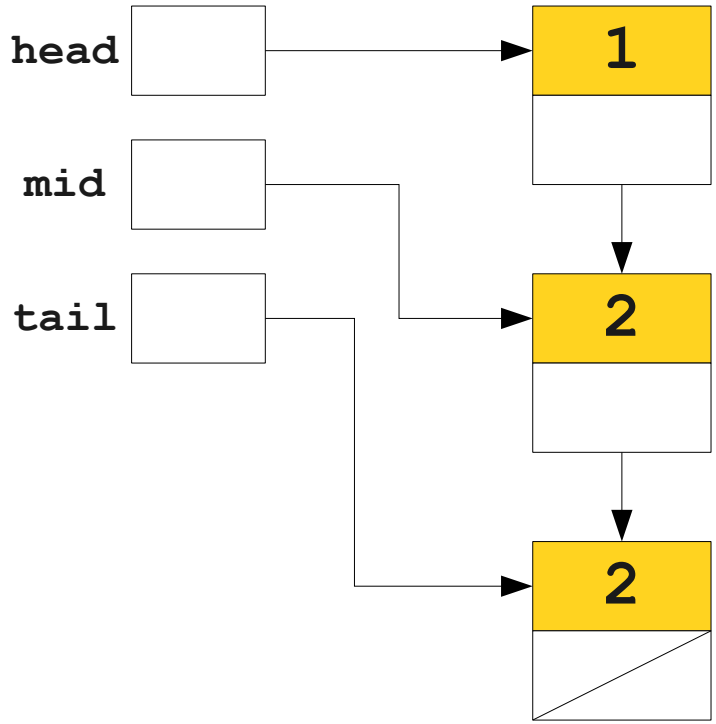
- Allocation is still done with malloc or object constructors but memory is automatically reclaimed
  - Data/Objects that won't be used again are called garbage
  - Reclaiming garbage objects automatically is called garbage collection
- Advantages:
  - Programmer does not have to worry about freeing unused resources
- Limitations:
  - Programmer can't reclaim unused resources
  - Difficult to implement and add a significant overhead

Implementation 1: reference counting

- Idea: if no pointer to a block exists, the block can safely be freed
- Add an extra field in each memory block (of the free list) with a count of the incoming pointers
  - When creating an object, set its counter to 0
  - When creating a reference to an object, increment its counter
  - When removing a reference, decrement its counter.
  - If zero, remove all outgoing references from that object and reclaim the memory

Reference counting: illustration

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    ....................................
    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



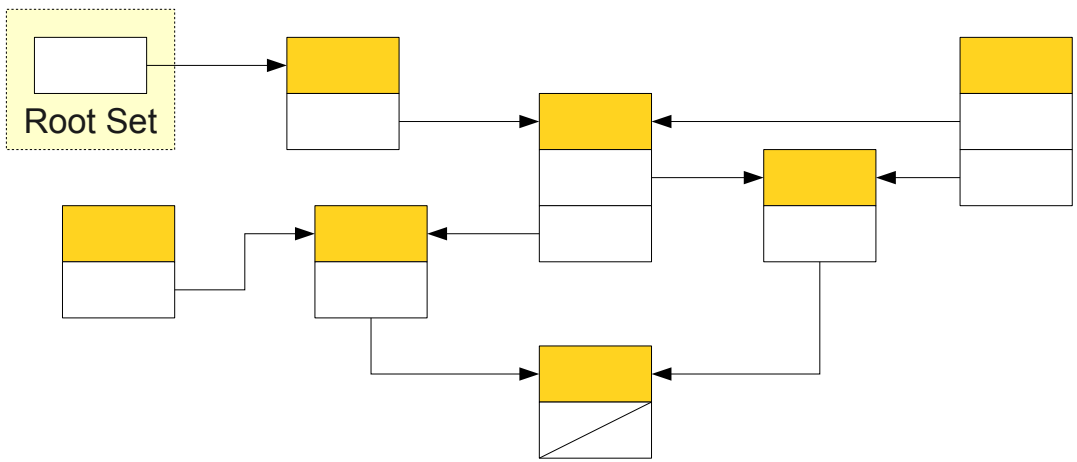
(Keith Schwarz)

Reference counting

- Straightforward to implement and can be combined with manual memory management
- Significant overhead when doing assignments for incrementing counters
- Impose constraints on the language
  - No pointer to the middle of an object, should be able to distinguish pointers from integers...
- Circular data structures are problematic
  - Counters will never be zero
  - E.g., doubly-linked lists
  - Algorithmic solutions exist but they are complex and costly.

Implementation 2: tracing garbage collectors

- Idea: find all reachable blocks from the knowledge of what is immediately accessible (the root set) and free all other blocks
- The root set is the set of memory locations that are known to be reachable
  - all variables in the program: registers, stack-allocated, global variables...
- Any objects (resp. not) reachable from the root set are (resp. not) reachable



Tracing garbage collection: mark-and-sweep

- Mark-and-sweep garbage collection:
  - Add a flag to each block
  - Marking phase: go through the graph, e.g., depth-first, setting the flag for all reached blocks
  - Sweeping phase: go through the list of blocks and free all unflagged ones
- Implementation of the mark stage with a stack:
  - Initialized to the root set
  - Retaining reachable blocks that have not yet been visited
- Tracing GC is typically called only when a malloc fails to avoid pauses in the program

- Problem: stack requires memory (and a malloc has just failed)
  - Marking phase can be implemented without a stack (at the expense of computing times)
  - Typically by adding descriptors within blocks and using *pointer reversal*

Implementation: tracing garbage collection

- Advantage:
  - More precise than reference counting
  - No problem with circular references
  - Run time can be made proportional to the number of reachable objects (typically much lower than number of free blocks)
- Disadvantages:
  - Introduce huge pause times
  - Consume lots of memory

Garbage collection

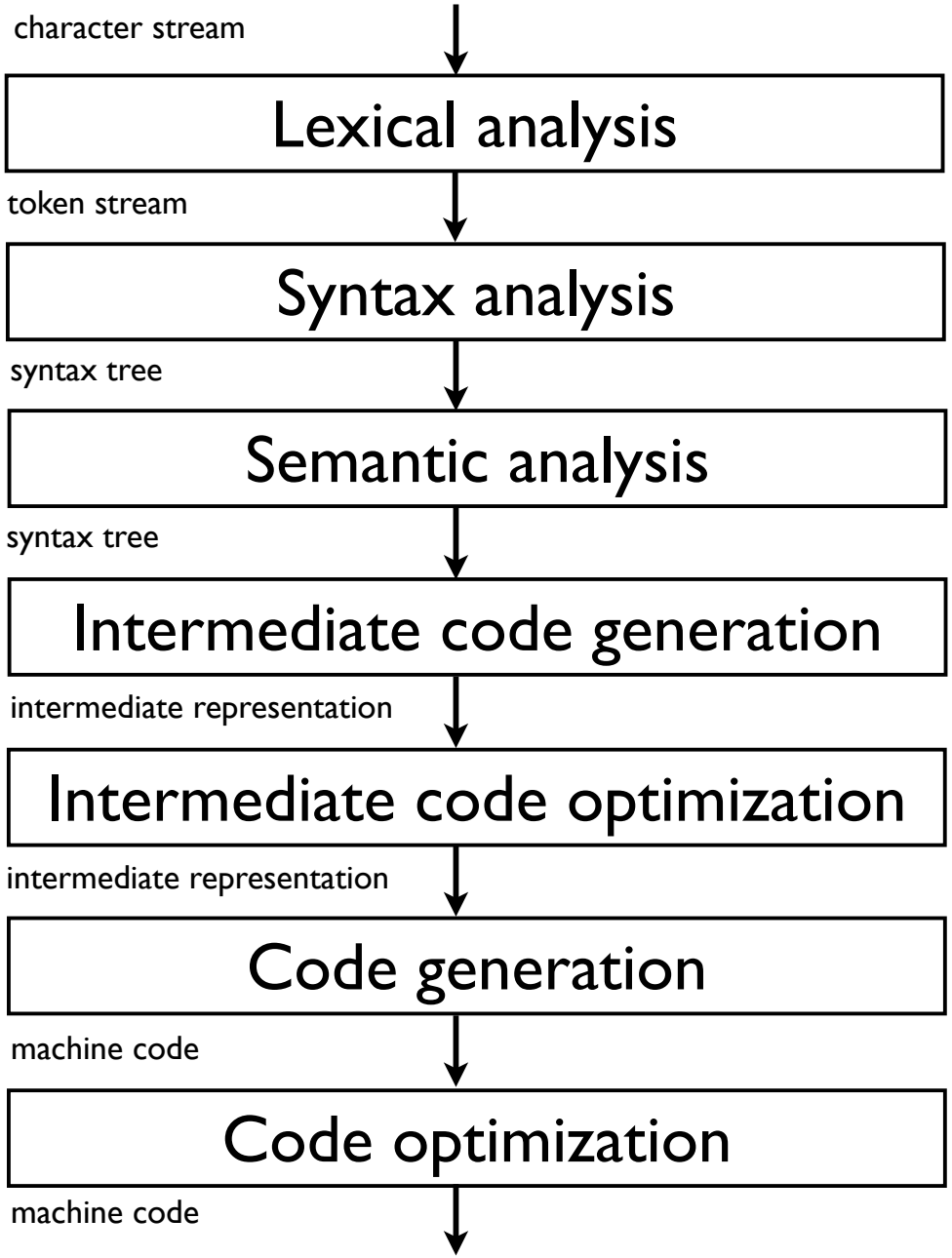
Other garbage collection methods:

- Two-space collection (stop-and-copying):
  - Avoid fragmentation and makes collection time proportional only to reachable nodes.
  - Two allocation spaces of same size are maintained
  - Blocks are always allocated in one space until full
  - Garbage collection then copies all live objects to the other space and swap their roles
- Generational collection:
  - Maintain several spaces for different generations of objects, with these spaces of increasing sizes
  - Optimized according to the "objects die young" principle
- Concurrent and incremental collectors
  - Perform collection incrementally or concurrently during execution of the program
  - Avoid long pauses but can reduce the total throughput

Part 7

Conclusion

Structure of a compiler



Summary

- Part 1, Introduction:
  - Overview and motivation...
- Part 2, Lexical analysis:
  - Regular expression, finite automata, implementation, Flex...
- Part 3, Syntax analysis:
  - Context-free grammar, top-down (predictive) parsing, bottom-up parsing (SLR and operator precedence parsing)...
- Part 4, Semantic analysis:
  - Syntax-directed translation, abstract syntax tree, type and scope checking...
- Part 5, Intermediate code generation and optimization:
  - Intermediate representations, IR code generation, optimization...
- Part 6, Code generation:
  - Instruction selection, register allocation, liveliness analysis, memory management...

More on compilers

- Our treatment of each compiler stage was superficial
- See reference books for more details (Transp. 4)

- Some things we have not discussed at all:
  - Specificities of object-oriented or functional programming languages
  - Machine dependent code optimization
  - Parallelism
  - ...

- Related topics:
  - Natural language processing
  - Domain-specific languages
  - ...