Quantum Compilers Week 4: Lexical and Syntax Analysis, with a Taste of Interpretation

(Including slides by J. Berthold and C. Oancea)

Department of Computer Science (DIKU)
University of Copenhagen

Spring 2024 02196 Quantum Compilers Lecture Slides

Fundamental Language Concepts

Lexicography: What is the alphabet and how do they form words? ('symbolic tokens' in formal languages).

Syntax: How do they form grammatical structures?

Semantics: What do they mean?

Fundamental Language Concepts

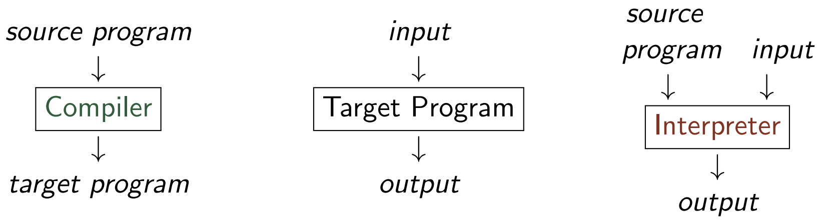
Lexicography: What is the alphabet and how do they form words? ('symbolic tokens' in formal languages).

Syntax: How do they form grammatical structures?

Semantics: What do they mean?

When designing a language, we must 1) define a *syntax* that is pleasant to work with as humans yet simple to represent structurally, and 2) assign a mapping from each *syntactical construct* to its well-defined *semantics*.

Compilation and Interpretation



Compilation and Interpretation

$$\begin{array}{c} \textit{source program} \\ \downarrow \\ \boxed{\text{Compiler}} \\ \downarrow \\ \textit{target program} \end{array} \qquad \begin{array}{c} \textit{input} \\ \downarrow \\ \boxed{\text{Target Program}} \\ \downarrow \\ \textit{output} \end{array} \qquad \begin{array}{c} \begin{array}{cc} \textit{source program} & \textit{input} \\ \downarrow & \downarrow \end{array} \\ \boxed{\text{Interpreter}} \\ \downarrow \\ \textit{output} \end{array}$$

- Compilation results in a lower-level language program, e.g., machine code, which can be run on various inputs. If the target language is at the same level, we call it a *transpiler*. In Danish we call both a “oversætter” (translator).
- Interpretation is good for impatient people:
  directly *executes* one by one the semantics of each syntactical construct in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.

Structure of a Classical Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree → Interpretation

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

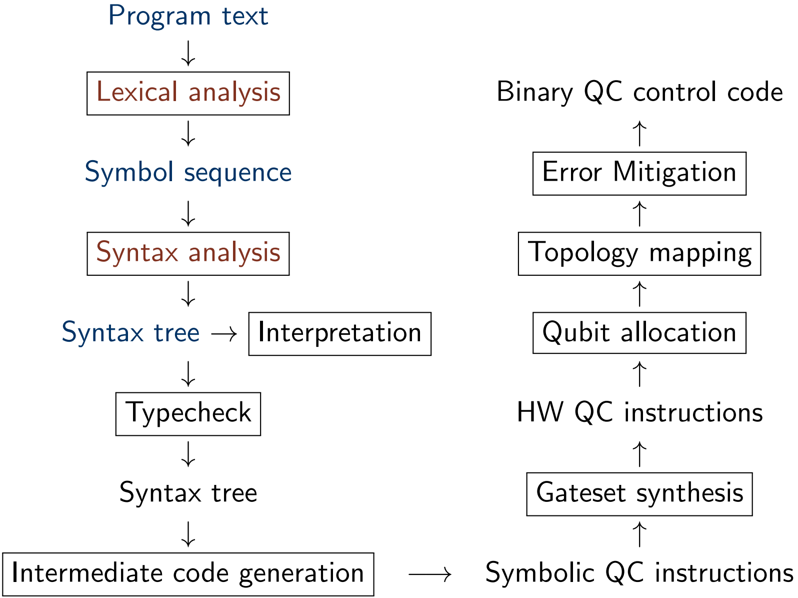
↑

Assembly and linking

↑

Binary machine code

Structure of a Quantum Compiler



Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":
: From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":
: From stream of symbolic token to abstract syntax tree (AST).

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

```
w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
```

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  ```
  w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
  ```

- Lexer produces:

  KEYWORD('while')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":
From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":
From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  ```
  w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
  ```

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":
From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":
From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

```
w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
```

- Lexer produces:

KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less')

Today’s topics: Lexical and Syntactical Analysis

Lexical analysis / ”Lexing”:

From stream of character to stream of symbolic tokens.

Syntactical analysis / ”Parsing”:

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

```
w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
```

- Lexer produces:

  KEYWORD(’while’) LPAR ID(’fun’) RPAR ID(’var’) OP(’less’) INT(12)

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

```
w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
```

- Lexer produces:

KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE ID('var')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  ```
  w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }
  ```

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE ID('var') OP('eq')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE ID('var') OP('eq') INT(2)

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE ID('var') OP('eq') INT(2) OP('multiply')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":

From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":

From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:

  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`

- Lexer produces:

  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE ID('var') OP('eq') INT(2) OP('multiply') ID('var')

Today's topics: Lexical and Syntactical Analysis

Lexical analysis / "Lexing":
From stream of character to stream of symbolic tokens.

Syntactical analysis / "Parsing":
From stream of symbolic token to abstract syntax tree (AST).

Example:

- input:
  `w h i l e ( f u n ( v a r ) < 1 2 ) { v a r = 2 * var ; }`
- Lexer produces:
  KEYWORD('while') LPAR ID('fun') RPAR ID('var') OP('less') INT(12) RPAR LBRACE ID('var') OP('eq') INT(2) OP('multiply') ID('var')
- Parser produces:
  Draw on blackboard.

1. Lexical Analysis; Regular Expressions

2. Syntax Analysis; Context-Free Grammars

Lexing: From character stream to token squence

Tokens can be e.g.:

- (fixed) vocabulary words, e.g., keywords (`let`, `if`, `then`, `else`, . . .), built-in operators (`*`, `::`), special symbols (`[`, `]`).

- Identifiers and Number Literals are **classes** of tokens, which are formed compositionally according to certain rules.

Formalism

Definition (Formal Languages)

Let Σ be an alphabet, i.e., a finite set of allowed characters.

- **A word** *over Σ is a string of chars $w = a_1 a_2 \ldots a_n$, $a_i \in \Sigma$*
  $n = 0$ is allowed and results in the empty string, denoted ϵ.
  Σ^* is the set of all words over Σ.
- **A language** *L over Σ is a set of words over Σ, i.e., $L \subseteq \Sigma^*$.*

Examples over the alphabet of lowercase Latin letters:

- Σ^* and $\emptyset$
- All C keywords: {`auto`, `break`, `case`, `...`, `while`}
- $\{a^n b^n \mid n \geq 0\}$
- All palindromes: {`kayak`, `racecar`, `mellem`, `retter`, `...`}
- $\{a^n b^n c^n \mid n \geq 0\}$

Formal languages encountered in compiler

Aim of compiler's front end: decide whether a program respects the object-language rules.

- Lexical analysis: decides whether the individual tokens are well formed. Requires the recognition of a simple language.
- Syntactical analysis: decides whether the composition of tokens is well formed: more complex language that checks compliance to grammar rules.
- Type checking: verifies that the program complies with (some of) the object language's typing rules. **Very** complex (but still effectively decidable) language

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, or `8`; but not `08` or `iv`
- Integers in hexadecimal format: `0X0123`, `0xcafe`; not `0X` or `00Xa`
- Floating point decimals: `0.`, `.345`, or `123.45`; not `3`, `.` or `0.1.2`
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`, ...

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, or `8`; but not `08` or `iv`
- Integers in hexadecimal format: `0X0123`, `0xcafe`; not `0X` or `00Xa`
- Floating point decimals: `0.`, `.345`, or `123.45`; not `3`, `.` or `0.1.2`
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`, ...

- A decimal integer is either `0` or a non-empty sequence of digits (`0-9`) that does not start with `0`.
- A hexadecimal integer starts with `0x` or `0X` and is followed by one or more hexadecimal digits (`0-9` or `a-f` or `A-F`).

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, or `8`; but not `08` or `iv`
- Integers in hexadecimal format: `0X0123`, `0xcafe`; not `0X` or `00Xa`
- Floating point decimals: `0.`, `.345`, or `123.45`; not `3`, `.` or `0.1.2`
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`, ...

- A decimal integer is either `0` or a non-empty sequence of digits (`0-9`) that does not start with `0`.
- A hexadecimal integer starts with `0x` or `0X` and is followed by one or more hexadecimal digits (`0-9` or `a-f` or `A-F`).
- Floating-point csts have a "mantissa," [...and] an "exponent," [...]. The mantissa is as a sequence of digits followed by a period, followed by an optional sequence of digits[...]. The exponent, if present, specifies the magnitude[...] using e or E[...] followed by an optional sign (+ or –) and a sequence of digits. If an exponent is present, the trailing decimal point is unnecessary in whole numbers. http://msdn.microsoft.com/en-us/library/tfh6f0w2.aspx.

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\Sigma)$ of regular expressions over alphabet Σ is defined:

- *Base Rules (Non Recursive):*
  - $\epsilon \in RE(\Sigma)$ *describes the lang consisting of only the empty string.*

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\Sigma)$ of regular expressions over alphabet Σ is defined:

- *Base Rules (Non Recursive):*
    - $\epsilon \in RE(\Sigma)$ *describes the lang consisting of only the empty string.*
    - $a \in RE(\Sigma)$, *for* $a \in \Sigma$, *describes the lang. of one-letter word* a.
- *Recursive Rules: for every* $r, s \in RE(\Sigma)$,

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\Sigma)$ of regular expressions over alphabet Σ is defined:

- *Base Rules (Non Recursive):*
  - $\epsilon \in RE(\Sigma)$ *describes the lang consisting of only the empty string.*
  - $a \in RE(\Sigma)$*, for* $a \in \Sigma$*, describes the lang. of one-letter word* a*.*
- *Recursive Rules: for every* $r, s \in RE(\Sigma)$*,*
  - $r \cdot s \in RE(\Sigma)$*, sequence/concatenation: words in which first part is described by* r*, and second part by* s*.*

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\Sigma)$ of regular expressions over alphabet Σ is defined:

- *Base Rules (Non Recursive):*
  - *$\epsilon \in RE(\Sigma)$ describes the lang consisting of only the empty string.*
  - *$a \in RE(\Sigma)$, for $a \in \Sigma$, describes the lang. of one-letter word a.*
- *Recursive Rules: for every $r, s \in RE(\Sigma)$,*
  - *$r \cdot s \in RE(\Sigma)$, sequence/concatenation: words in which first part is described by r, and second part by s.*
  - *$r \mid s \in RE(\Sigma)$, alternative/union: words described by r OR by s.*

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\Sigma)$ of regular expressions over alphabet Σ is defined:

- *Base Rules (Non Recursive):*
  - *$\epsilon \in RE(\Sigma)$ describes the lang consisting of only the empty string.*
  - *$a \in RE(\Sigma)$, for $a \in \Sigma$, describes the lang. of one-letter word a.*
- *Recursive Rules: for every $r, s \in RE(\Sigma)$,*
  - *$r \cdot s \in RE(\Sigma)$, sequence/concatenation: words in which first part is described by r, and second part by s.*
  - *$r \mid s \in RE(\Sigma)$, alternative/union: words described by r OR by s.*
  - *$r^* \in RE(\Sigma)$, repetition: zero or more words described by r.*

- We may use parentheses `(...)` for grouping regular expressions.
- We will often omit the explicit $\cdot$ in sequences.
- Sequence groups tighter than alternative: $\texttt{a|bc}^* = \texttt{a|(b(c}^*\texttt{))}$.

Demonstrating Regular-Expression Combinators

$r \cdot s$ Assume the languages of regular expressions r and s are L(r)={"a","b"} and L(s)={"c","de"}, respectively.

Then L($r \cdot s$)=

Demonstrating Regular-Expression Combinators

$r \cdot s$ Assume the languages of regular expressions r and s are `L(`r`)={"a","b"}` and `L(`s`)={"c","de"}`, respectively.

Then `L(`$r \cdot s$`)={"ac", "ade", "bc", "bde"}`.

When matching keywords, `if` is the concatenation of two regular expressions: `i` and `f`.

r^* Assume the language of regular expression r is `L(`r`)={"a","bb"}`.

Then

`L(`r^*`)=`

Demonstrating Regular-Expression Combinators

$r \cdot s$ Assume the languages of regular expressions r and s are L(r)={"a","b"} and L(s)={"c","de"}, respectively.

Then L($r \cdot s$)={"ac", "ade", "bc", "bde"}.

When matching keywords, `if` is the concatenation of two regular expressions: `i` and `f`.

r^* Assume the language of regular expression r is L(r)={"a","bb"}.

Then

L(r^*)={"","a","bb","aa","abb","bba","bbbb","aaa",...}.

Examples: Integers and Variable Names in C++

- Integers in decimal format: `234`, `0`, `8`; but not `08` or `iv`
- Integers in hexadecimal format: `0X0123`, `0xcafe`; not `0X`, `00Xa`
- A variable name consists of letters, digits and underscore, and it must begin with a letter or underscore.

Examples: Integers and Variable Names in C++

- Integers in decimal format: `234`, `0`, `8`; but not `08` or `iv`
- Integers in hexadecimal format: `0X0123`, `0xcafe`; not `0X`, `00Xa`
- A variable name consists of letters, digits and underscore, and it must begin with a letter or underscore.

- Integers in decimal format:
  `(1|2|···|9)(0|1|2|···|9)* | 0`
  Shorthand via "character range" (`[-]`): `[1-9][0-9]*|0`
- Integers in hexadecimal format:
  `0(x|X)[0-9a-fA-F][0-9a-fA-F]*`
  Shorthand via "at least one" (`+`): `0(x|X)[0-9a-fA-F]+`.
- Variable names: `[a-zA-Z_][a-zA-Z_0-9]*`

Useful Abbreviations for Regular Expressions

- Character Sets: $[\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n] := (\mathtt{a}_1 \mid \mathtt{a}_2 \mid \ldots \mid \mathtt{a}_n)$, i.e., one of $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \in \Sigma$.
- Negated Character Sets: $[\hat{}\,\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n]$ describes any $\mathtt{a} \in \Sigma \setminus \{\mathtt{a}_1,\mathtt{a}_2,\ldots,\mathtt{a}_n\}$.
- Character Ranges: $[\mathtt{a}_1\text{-}\mathtt{a}_n] := (\mathtt{a}_1 \mid \mathtt{a}_2 \mid \cdots \mid \mathtt{a}_n)$, where $\{\mathtt{a}_i\}$ is ordered, i.e., one character in the range $\mathtt{a}_1$ through $\mathtt{a}_n$.
- Optional Parts: $r? := (r \mid \epsilon)$ for $r \in RE(\Sigma)$, optionally a string described by r.
- Repeated Parts: $r^+ := (r\,r^*)$ for $r \in RE(\Sigma)$, *at least one* string described by r (but possibly more).

Properties of Regular Expression Combinators

- $|$ is associative: $(r\,|\,s)\,|\,t \;=\; r\,|\,(s\,|\,t) \;=\; r\,|\,s\,|\,t$
- $|$ is commutative: $s\,|\,t \;=\; t\,|\,s$
- $|$ is idempotent: $s\,|\,s \;=\; s$
- Also, by definition: $s? \;=\; s\,|\,\epsilon$

- $\cdot$ is associative: $(r\,s)\,t \;=\; r\,(s\,t) \;=\; r\,s\,t$
- ϵ is neutral element for $\cdot$: $s\,\epsilon \;=\; \epsilon\,s = s$
- $\cdot$ distributes over $|$: $r\,(s\,|\,t) \;=\; r\,s\,|\,r\,t$, and $(r\,|\,s)\,t \;=\; r\,t\,|\,s\,t$.

- * is idempotent: $(s^*)^* = s^*$.
- Also, $s^*s^* = s^*$, and $s\,s^* = s^+ = s^*s$ by definition!

In all of these, $=$ means "describes the same language as".

Lexer generators

The lexer generator uses compositionality of regular expressions to build one large state machine that recognizes *all* tokens extremely efficiently, using only one lookup per input character.

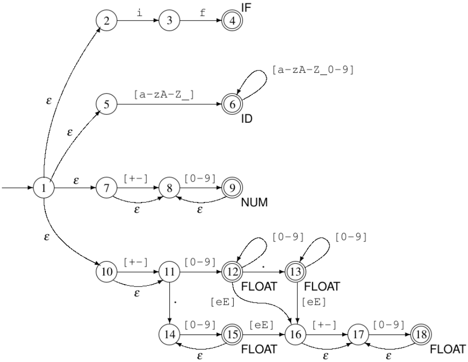


Figure: Combined NFA for 4 tokens

Lexer generators

The lexer generator uses compositionality of regular expressions to build one large state machine that recognizes *all* tokens extremely efficiently, using only one lookup per input character.

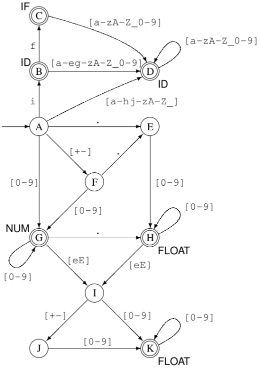


Figure: Minimized DFA for the 4-token NFA

1. Lexical Analysis; Regular Expressions

2. Syntax Analysis; Context-Free Grammars

Syntax Analysis (Parsing)

Relates to the correct construction of sentences, i.e., grammar.

1. Checks that grammar is respected, otherwise syntax error, and
2. Arranges tokens into a syntax tree reflecting the text structure: leaves are tokens, which if read from left to right results in the original text!

Essential tool and theory used are *Context-Free Grammars*: a notation suitable for human understanding that can be transformed into an efficient implementation.

Context-Free Grammar (CFG) Definition

1. a set of *terminals* Σ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: lowercase letters, Lark: uppercase.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings. (Convention: uppercase letters, Lark: lowercase.)
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \rightarrow X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\Sigma \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.
   May abbreviate productions $Y \rightarrow \vec{X}_1, Y \rightarrow \vec{X}_2$ as $Y \rightarrow \vec{X}_1 \mid \vec{X}_2$.

G: $S \rightarrow aS$
$\quad S \rightarrow \epsilon$

Context-Free Grammar (CFG) Definition

1 a set of *terminals* Σ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: lowercase letters, Lark: uppercase.)

2 a set of *non-terminals* N, denoting sets of recursively defined strings. (Convention: uppercase letters, Lark: lowercase.)

3 *a start symbol* $S \in N$, denoting the lang defined by the grammar.

4 a set P of productions of form $Y \to X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\Sigma \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.
May abbreviate productions $Y \to \vec{X}_1, Y \to \vec{X}_2$ as $Y \to \vec{X}_1 \mid \vec{X}_2$.

G: $S \to aS$
$S \to \epsilon$
regular-expression language a^*

G: $S \to aSb$
$S \to \epsilon$

Context-Free Grammar (CFG) Definition

1. a set of *terminals* Σ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: lowercase letters, Lark: uppercase.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings. (Convention: uppercase letters, Lark: lowercase.)
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \to X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\Sigma \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.
   May abbreviate productions $Y \to \vec{X}_1, Y \to \vec{X}_2$ as $Y \to \vec{X}_1 \mid \vec{X}_2$.

G: $S \to aS$
$S \to \epsilon$
regular-expression language a^*

G: $S \to aSb$
$S \to \epsilon$
describes language $\{a^n b^n, \forall n \geq 0\}$

G: $S \to aSa \mid bSb \mid \ldots$
$S \to a \mid b \mid \ldots \mid \epsilon$

Context-Free Grammar (CFG) Definition

1. a set of *terminals* Σ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: lowercase letters, Lark: uppercase.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings. (Convention: uppercase letters, Lark: lowercase.)
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \rightarrow X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\Sigma \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.
   May abbreviate productions $Y \rightarrow \vec{X}_1, Y \rightarrow \vec{X}_2$ as $Y \rightarrow \vec{X}_1 \mid \vec{X}_2$.

G: $S \rightarrow aS$
$S \rightarrow \epsilon$
regular-expression language a^*

G: $S \rightarrow aSb$
$S \rightarrow \epsilon$
describes language $\{a^n b^n, \forall n \geq 0\}$

G: $S \rightarrow aSa \mid bSb \mid \ \ldots$
$S \rightarrow a \mid b \mid \ \ldots \ \mid \epsilon$
describes palindromes, e.g., *abba*, *babab*.

The latter two languages cannot be described with regular expressions.

Example: Deriving Words

Nonteminals recursively refer to themselves or each other (cannot do that with regular expressions):

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)

G: $S \rightarrow aSB \mid \epsilon$ $\quad S = \{a \cdot x \cdot y \mid x \in S,\ y \in B\} \cup \{\epsilon\}$
$B \rightarrow Bb \mid b$ $\quad B = \{x \cdot b \mid x \in B\} \cup \{b\}$

'Sentences' in the language can be constructed by

- starting with the start symbol S, and
- successively replacing nonterminals with right-hand sides.

Deriving *aaabbbb* (each step replaces $\underline{Y}$ on LHS with $\overline{X_1 ... X_n}$):

Example: Deriving Words

Nonteminals recursively refer to themselves or each other (cannot do that with regular expressions):

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)

G: $S \rightarrow aSB \mid \epsilon$
$B \rightarrow Bb \mid b$

$S = \{a \cdot x \cdot y \mid x \in S,\ y \in B\} \cup \{\epsilon\}$
$B = \{x \cdot b \mid x \in B\} \cup \{b\}$

'Sentences' in the language can be constructed by

- starting with the start symbol S, and
- successively replacing nonterminals with right-hand sides.

Deriving *aaabbbb* (each step replaces $\underline{Y}$ on LHS with $\overline{X_1 ... X_n}$):

$$\underline{S} \Rightarrow^1 \overline{a\underline{S}B} \Rightarrow^1 a\overline{aS\underline{B}}B \Rightarrow^4 aa\underline{S}\overline{b}B \Rightarrow^1 aa\overline{a\underline{S}B}bB$$

Example: Deriving Words

Nonteminals recursively refer to themselves or each other (cannot do that with regular expressions):

$$
\begin{array}{lll}
\text{G: } S \to aSB & (1) \\
\quad S \to \epsilon & (2) \\
\quad B \to Bb & (3) \\
\quad B \to b & (4)
\end{array}
\qquad
\begin{array}{l}
\text{G: } S \to aSB \mid \epsilon \\
\quad B \to Bb \mid b
\end{array}
\qquad
\begin{array}{l}
S = \{a\cdot x\cdot y \mid x \in S,\ y \in B\} \cup \{\epsilon\} \\
B = \{x \cdot b \mid x \in B\} \cup \{b\}
\end{array}
$$

'Sentences' in the language can be constructed by

- starting with the start symbol S, and
- successively replacing nonterminals with right-hand sides.

Deriving *aaabbbb* (each step replaces $\underline{Y}$ on LHS with $\overline{X_1 ... X_n}$):

$$
\begin{array}{l}
\underline{S} \Rightarrow^1 \overline{a\underline{S}B} \Rightarrow^1 a\overline{aS\underline{B}}B \Rightarrow^4 aa\underline{S}\overline{b}B \Rightarrow^1 aa\overline{a\underline{S}B}bB \\
\quad \Rightarrow^2 aaa\overline{}\underline{B}bB \Rightarrow^3 aaa\overline{Bb}b\underline{B} \Rightarrow^4 aaa\underline{B}bb\overline{b} \Rightarrow^4 aaa\overline{b}bbb
\end{array}
$$

Definition: Derivation Relation

Let $G = (\Sigma, N, S, P)$ be a grammar.
The derivation relation $\Rightarrow$ on $(\Sigma \cup N)^*$ is defined as:

- For a nonterminal $X \in N$ and a production $(X \to \beta) \in P$, $\alpha_1 X \alpha_2 \;\Rightarrow\; \alpha_1 \beta \alpha_2$, for all $\alpha_1, \alpha_2 \in (\Sigma \cup N)^*$
- Describes one derivation step using one of the productions.
- Each step may be optionally annotated with the grammar-rule number.

G:
$$
\begin{array}{ll}
S \to aSB & (1) \\
S \to \epsilon & (2) \\
B \to Bb & (3) \\
B \to b & (4)
\end{array}
$$

$$S \;\Rightarrow^1\; aSB \;\Rightarrow^1\; aaSBB \;\Rightarrow^2\; aa\,BB$$

Definition: Derivation Relation

Let $G = (\Sigma, N, S, P)$ be a grammar.
The derivation relation $\Rightarrow$ on $(\Sigma \cup N)^*$ is defined as:

- For a nonterminal $X \in N$ and a production $(X \to \beta) \in P$, $\alpha_1 X \alpha_2 \;\Rightarrow\; \alpha_1 \beta \alpha_2$, for all $\alpha_1, \alpha_2 \in (\Sigma \cup N)^*$
- Describes one derivation step using one of the productions.
- Each step may be optionally annotated with the grammar-rule number.

G: $S \to aSB$ (1)
$S \to \epsilon$ (2)
$B \to Bb$ (3)
$B \to b$ (4)

$$S \Rightarrow^1 aSB \Rightarrow^1 aaSBB \Rightarrow^2 aa\ BB$$
$$\Rightarrow^3 aaBbB \Rightarrow^4 aabbB \Rightarrow^4 aabbb.$$

- Here we have used leftmost derivation, i.e., always expanded the leftmost terminal first. Could also use right-most derivation.
- $aaabbbb$ and $aabbb \in L(G)$.

Transitive Derivation Relation Definition

Let $G = (\Sigma, N, S, P)$ be a grammar and $\Rightarrow$ its derivation relation. The transitive derivation relation $\Rightarrow^*$ is defined as:

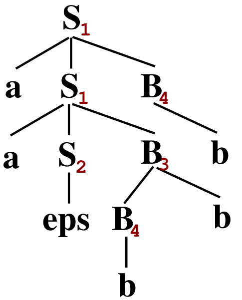
- $\alpha \Rightarrow^* \alpha$, for $\alpha \in (\Sigma \cup N)^*$, derived in 0 steps,
- for $\alpha, \beta \in (\Sigma \cup N)^*$, $\alpha \Rightarrow^* \beta$ *iff* there exists $\gamma \in (\Sigma \cup N)^*$ such that $\alpha \Rightarrow \gamma$, and $\gamma \Rightarrow^* \beta$, i.e., derived in at least one step.

The Language of a Grammar consists of all the words that can be obtained via the transitive derivation relation:
$L(G) = \{w \in \Sigma^* \mid S \ \Rightarrow^* \ w\}$.

For example *aaabbbb* and *aabbb* $\in L(G)$,
because $S \ \Rightarrow^* \ aaabbbb$ and $S \ \Rightarrow^* \ aabbb$.

Syntax Trees

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)



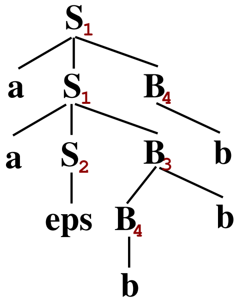
Syntax trees describe the "structure" of the derivation (independent of the order in which nonterminals have been chosen to be derived).

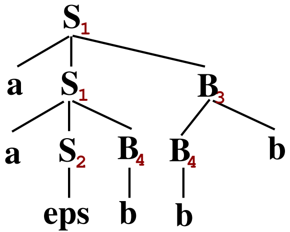
Leftmost derivation always derives the leftmost nonterminal first, and corresponds to a *depth-first, left-to-right, preorder tree traversal*:

$$\underline{S} \Rightarrow^1 \overline{a\underline{S}B} \Rightarrow^1 a\overline{a\underline{S}B}B \Rightarrow^2 aa\,\underline{B}B \Rightarrow^3 aa\overline{\underline{B}b}B \Rightarrow^4 aa\bar{b}b\underline{B} \Rightarrow^4 aabb\bar{b}.$$

Syntax Trees & Ambiguous Grammars

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)





Syntax trees describe the "structure" of the derivation (independent of the order in which nonterminals have been chosen to be derived).

The grammar is said to be ambiguous if there exists a word that can be derived in two ways, corresponding to different syntax trees.

$\underline{S} \Rightarrow^1 \overline{a\underline{S}B} \Rightarrow^1 a\overline{a\underline{S}B}B \Rightarrow^2 aa\underline{B}B \Rightarrow^3 aa\overline{\underline{B}b}B \Rightarrow^4 aa\overline{b}b\underline{B} \Rightarrow^4 aabb\overline{b}.$
$\underline{S} \Rightarrow^1 \overline{a\underline{S}B} \Rightarrow^1 a\overline{a\underline{S}B}B \Rightarrow^2 aa\underline{B}B \Rightarrow^4 aa\overline{b}\underline{B} \Rightarrow^3 aab\overline{\underline{B}b} \Rightarrow^4 aabb\overline{b}.$

Handling/Removing Grammar Ambiguity

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \\ E & \rightarrow & E * E \mid E \,/\, E \\ E & \rightarrow & a \mid (E) \end{array}$$

- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a + a * a$

Handling/Removing Grammar Ambiguity

$$
\begin{array}{lll}
E & \rightarrow & E + E \mid E - E \\
E & \rightarrow & E * E \mid E \;/\; E \\
E & \rightarrow & a \mid (E)
\end{array}
$$

- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a + a * a$ can be resolved by setting *the precedence* of $*$ higher than $+$: $a + (a * a)$.
- Ambiguous derivation of $a - a - a$

Handling/Removing Grammar Ambiguity

$$
\begin{array}{lcl}
E & \rightarrow & E + E \mid E - E \\
E & \rightarrow & E * E \mid E \,/\, E \\
E & \rightarrow & a \mid (E)
\end{array}
$$

- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a + a * a$ can be resolved by setting *the precedence* of $*$ higher than $+$: $a + (a * a)$.
- Ambiguous derivation of $a - a - a$ can be resolved by fixing *a left-associative* derivation: $(a - a) - a$.

Defining/Resolving Operator Precedence

- Introduce precedence levels to set operator priorities
- for example precedence of $*$ and $/$ over (higher than) $+$ and $-$,
- and more precedence levels can be added, e.g., exponentiation.

Defining/Resolving Operator Precedence

- Introduce precedence levels to set operator priorities
- for example precedence of $*$ and $/$ over (higher than) $+$ and $-$,
- and more precedence levels can be added, e.g., exponentiation.

At grammar level: this can be accomplished by introducing one nonterminal for each level of precedence:

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \\ E & \rightarrow & E * E \mid E \, / \, E \\ E & \rightarrow & a \mid (E) \end{array}$$

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T \, / \, T \\ T & \rightarrow & a \mid (E) \end{array}$$

Defining/Resolving Operator Associativity

A binary operator $\oplus$ is called:

- *left associative* if expression $x \oplus y \oplus z$ should be grouped from left to right: $(x \oplus y) \oplus z$
- *right associative* if expression $x \oplus y \oplus z$ should be grouped from right to left: $x \oplus (y \oplus z)$
- *non-associative* if expressions such as $x \oplus y \oplus z$ are disallowed,
- associative if both left-to-right and right-to-left groupings lead to the same result (a semantic, not syntactic, property).

Examples:

- left associative operators: $-$ and $/$,
- right associative operators

Defining/Resolving Operator Associativity

A binary operator $\oplus$ is called:

- *left associative* if expression $x \oplus y \oplus z$ should be grouped from left to right: $(x \oplus y) \oplus z$
- *right associative* if expression $x \oplus y \oplus z$ should be grouped from right to left: $x \oplus (y \oplus z)$
- *non-associative* if expressions such as $x \oplus y \oplus z$ are disallowed,
- associative if both left-to-right and right-to-left groupings lead to the same result (a semantic, not syntactic, property).

Examples:

- left associative operators: $-$ and $/$,
- right associative operators: exponentiation, assignment (in C, Java: `a = b = c`), arrow (in F# types: `int -> int -> int`).

Establishing Intended Associativity

- Can be declared in the parser file via directives
- when operators are semantically associative (e.g., +) , use same associativity as comparable operators (e.g., −)
- cannot mix left- and right-associative operators at the same precedence level.

Establishing Intended Associativity

- Can be declared in the parser file via directives
- when operators are semantically associative (e.g., +) , use same associativity as comparable operators (e.g., −)
- cannot mix left- and right-associative operators at the same precedence level.

At grammar level: this can be accomplished by introducing new nonterminals that establish explicitly operator's associativity :

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T \,/\, T \\ T & \rightarrow & a \mid (E) \end{array}$$

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T \,/\, F \mid F \\ F & \rightarrow & a \mid (E) \end{array}$$

Establishing Intended Associativity

- Can be declared in the parser file via directives
- when operators are semantically associative (e.g., +) , use same associativity as comparable operators (e.g., –)
- cannot mix left- and right-associative operators at the same precedence level.

At grammar level: this can be accomplished by introducing new nonterminals that establish explicitly operator's associativity :

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T \,/\, T \\ T & \rightarrow & a \mid (E) \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T \,/\, F \mid F \\ F & \rightarrow & a \mid (E) \end{array}$$

- Left associative $\Rightarrow$ Left-recursive grammar production.
- Right associative $\Rightarrow$ Right-recursive grammar production.