Bab 5

Context-Free Grammar dan Pengenalan Parsing

5.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep context-free grammar (CFG) dan perannya dalam syntax analysis
2. Menjelaskan notasi BNF (Backus-Naur Form) dan EBNF (Extended BNF)
3. Menulis grammar untuk ekspresi aritmatika dan konstruksi bahasa sederhana
4. Memahami konsep derivation (leftmost dan rightmost)
5. Membuat parse tree untuk kalimat yang diberikan
6. Mengidentifikasi dan menjelaskan ambiguity dalam grammar
7. Memahami hubungan antara grammar, parsing, dan syntax analysis

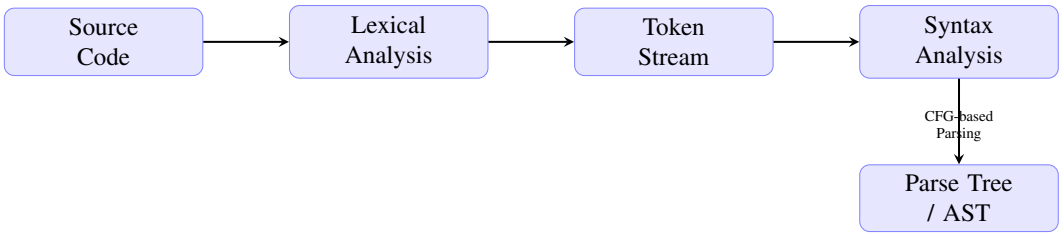
5.2 Pendahuluan

Setelah lexical analysis menghasilkan stream token, fase berikutnya dalam kompilator adalah syntax analysis atau parsing. Menurut sumber terbuka:

> "Given the stream of tokens from the lexer, syntax analysis checks whether they form a valid sequence under the language grammar. Builds a parse tree or AST that represents nested structure of language constructs."[1]

Syntax analysis membutuhkan formal grammar untuk mendefinisikan struktur yang valid dalam bahasa pemrograman. Context-free grammar (CFG) adalah alat formal yang paling umum digunakan untuk tujuan ini karena kemampuannya dalam menangani struktur nested dan recursive yang umum ditemui dalam bahasa pemrograman.

Gambar 5.1 menunjukkan posisi syntax analysis dalam pipeline kompilator.



Gambar 5.1: Posisi syntax analysis dalam pipeline kompilator

5.3 Grammar Proyek Subset C

Grammar lengkap proyek compiler subset C telah didefinisikan di Bab 1 (Bagian **??**). Berikut ringkasan dalam notasi BNF yang akan dipakai di Bab 6 (parser hand-written) dan Bab 8 (parser Bison):

- **program**: satu atau lebih statement.
- **statement**: declaration `;` atau assignment `;` atau print-statement `;`
- **declaration**: `int` identifier atau `float` identifier
- **assignment**: identifier = expr
- **print-statement**: `print (` string-literal `)` atau `print (` expr `)`
- **expr**: term, atau expr + term, atau expr − term (associativity kiri)
- **term**: factor, atau term `*` factor, atau term `/` factor (associativity kiri)
- **factor**: literal, atau identifier, atau `(` expr `)`

Precedence: `*` dan `/` lebih tinggi dari `+` dan `-`. Parser proyek di Bab 8 mengimplementasikan grammar ini dalam file `simplec.y`; lexer proyek di Bab 4 (`simplec.l`) menghasilkan token set yang sesuai dengan spesifikasi Bab 1.

5.4 Context-Free Grammar (CFG)

5.4.1 Definisi Formal

Context-free grammar adalah tipe formal grammar yang didefinisikan sebagai tuple $G = (V, \Sigma, R, S)$ dimana:

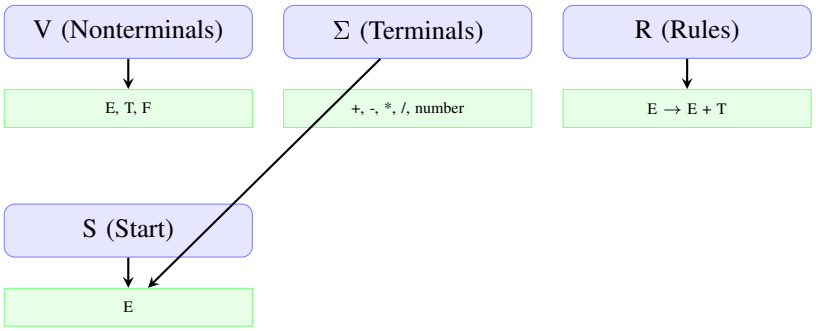
- V adalah himpunan **nonterminal symbols** (variabel yang dapat di-expand)
- Σ adalah himpunan **terminal symbols** (token yang tidak dapat di-expand, disjoint dari V)

- R adalah himpunan **productions** (rules) dengan bentuk $A \to \alpha$, dimana $A \in V$ dan $\alpha \in (V \cup \Sigma)^*$

- $S \in V$ adalah **start symbol**

CFG disebut "context-free" karena aturan produksi dapat diterapkan tanpa mempertimbangkan konteks di sekitar nonterminal. Artinya, jika ada produksi $A \to \alpha$, maka A dapat diganti dengan α di manapun A muncul, terlepas dari simbol di sekitarnya.

Gambar 5.2 menunjukkan komponen-komponen CFG secara visual.



Gambar 5.2: Komponen-komponen CFG: $G = (V, \Sigma, R, S)$

5.4.2 Contoh Grammar Sederhana

Mari kita lihat contoh grammar untuk ekspresi aritmatika sederhana:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

Dalam grammar ini:

- **Nonterminals**: E (expression), T (term), F (factor)

- **Terminals**: `+`, `-`, `*`, `/`, `(`, `)`, `number`

- **Start symbol**: E

Grammar ini dapat menghasilkan ekspresi seperti `3 + 4 * 5`, `(2 + 3) * 4`, dll.

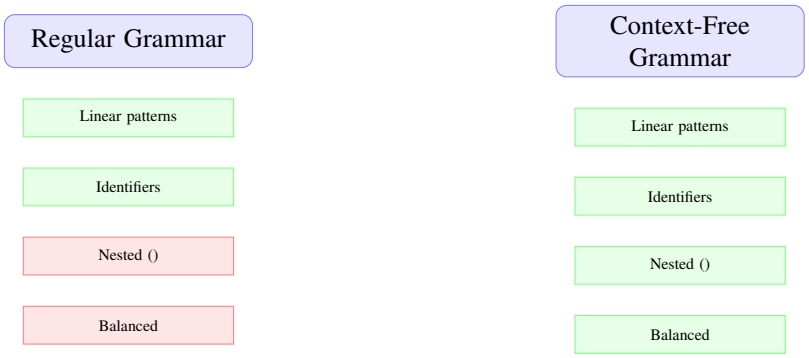
5.4.3 Perbedaan Regular Grammar dan Context-Free Grammar

Penting untuk memahami perbedaan antara regular grammar (yang digunakan untuk lexical analysis) dan context-free grammar:

- **Regular Grammar**: Hanya dapat menangani struktur linear, tidak dapat menangani nested structures seperti parentheses yang seimbang
- **Context-Free Grammar**: Dapat menangani struktur nested dan recursive, seperti:
  - Parentheses matching: `( ( ( ) ) )`
  - Nested blocks: `{ { } }`
  - Recursive function calls
  - Nested if-statements

Inilah mengapa CFG digunakan untuk syntax analysis, sementara regular grammar cukup untuk lexical analysis.

Gambar 5.3 menunjukkan perbandingan kemampuan regular grammar dan CFG.



Gambar 5.3: Perbandingan kemampuan Regular Grammar vs CFG

5.5 BNF dan EBNF Notasi

5.5.1 Backus-Naur Form (BNF)

BNF adalah notasi metalanguage yang dikembangkan oleh John Backus dan Peter Naur untuk mendefinisikan syntax bahasa pemrograman. BNF menggunakan simbol-simbol berikut:

- $::=$ atau $\rightarrow$: Menandakan "didefinisikan sebagai"
- $|$: Menandakan alternatif (OR)

- `<nonterminal>`: Nonterminal symbol (biasanya dalam angle brackets)
- `terminal`: Terminal symbol (biasanya tanpa angle brackets)

Contoh grammar dalam BNF:

```
<expression> ::= <expression> + <term>
               | <expression> - <term>
               | <term>

<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>

<factor> ::= ( <expression> )
           | <number>
```

5.5.2 Extended BNF (EBNF)

EBNF memperluas BNF dengan konstruksi tambahan untuk membuat grammar lebih kompak dan mudah dibaca:

- **Optional**: `[ ... ]` atau `?` - elemen opsional (nol atau satu kali)
- **Repetition**:
  - `*`: Nol atau lebih kali
  - `+`: Satu atau lebih kali
- **Grouping**: `( ... )` - untuk mengelompokkan
- **Terminal strings**: `"..."` atau `'...'` - untuk literal strings

Contoh grammar yang sama dalam EBNF:

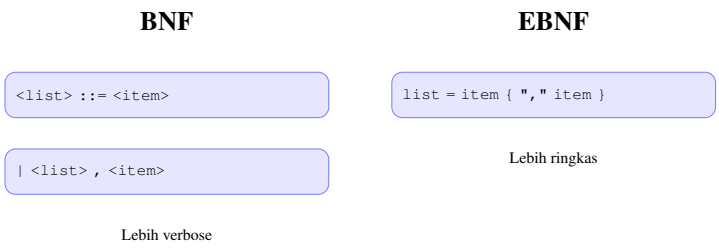
```
expression = term { ("+" | "-") term }
term       = factor { ("*" | "/") factor }
factor     = "(" expression ")" | number
```

EBNF lebih ringkas dan mudah dibaca. Banyak spesifikasi bahasa modern menggunakan EBNF, termasuk ISO standard untuk grammar notation.

Gambar 5.4 menunjukkan perbandingan notasi BNF dan EBNF.

5.5.3 Contoh Grammar untuk Konstruksi Bahasa

Berikut contoh grammar untuk beberapa konstruksi bahasa pemrograman:



Gambar 5.4: Perbandingan notasi BNF dan EBNF

If-Statement

```
<if_statement> ::= if ( <expression> ) <statement>
                 | if ( <expression> ) <statement> else <statement>
```

Dalam EBNF:

```
if_statement = "if" "(" expression ")" statement
               [ "else" statement ]
```

While Loop

```
<while_statement> ::= while ( <expression> ) <statement>
```

Dalam EBNF:

```
while_statement = "while" "(" expression ")" statement
```

Variable Declaration

```
<declaration> ::= <type> <identifier> [ = <expression> ] ;
```

Dalam EBNF:

```
declaration = type identifier [ "=" expression ] ";"
```

5.6 Derivation

Derivation adalah proses menerapkan aturan produksi untuk menghasilkan string terminal dari start symbol. Terdapat dua jenis derivation yang penting:

5.6.1 Leftmost Derivation

Leftmost derivation selalu mengganti nonterminal paling kiri terlebih dahulu pada setiap langkah.

Contoh untuk ekspresi `3 + 4 * 5` dengan grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

Leftmost derivation:

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow \texttt{3} + T \\
&\Rightarrow \texttt{3} + T * F \\
&\Rightarrow \texttt{3} + F * F \\
&\Rightarrow \texttt{3} + \texttt{4} * F \\
&\Rightarrow \texttt{3} + \texttt{4} * \texttt{5}
\end{aligned}
$$

5.6.2 Rightmost Derivation

Rightmost derivation selalu mengganti nonterminal paling kanan terlebih dahulu pada setiap langkah.

Rightmost derivation untuk `3 + 4 * 5`:

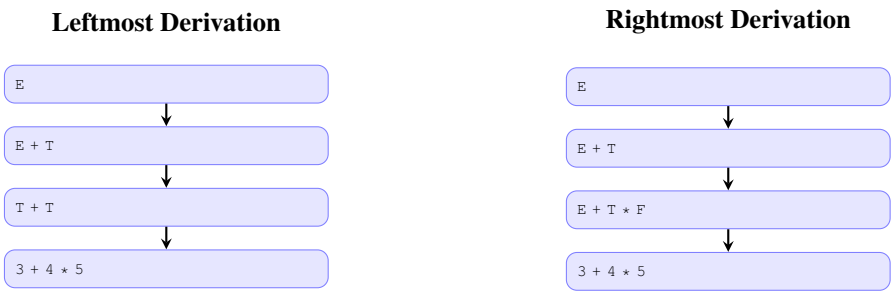
$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow E + T * F \\
&\Rightarrow E + T * \texttt{5} \\
&\Rightarrow E + F * \texttt{5} \\
&\Rightarrow E + \texttt{4} * \texttt{5} \\
&\Rightarrow T + \texttt{4} * \texttt{5} \\
&\Rightarrow F + \texttt{4} * \texttt{5} \\
&\Rightarrow \texttt{3} + \texttt{4} * \texttt{5}
\end{aligned}
$$

Gambar 5.5 menunjukkan perbandingan leftmost dan rightmost derivation.

5.6.3 Pentingnya Derivation

Derivation penting karena:

- Menunjukkan bagaimana string dihasilkan dari grammar
- Menentukan urutan penggantian nonterminal (penting untuk parsing)



Gambar 5.5: Perbandingan leftmost dan rightmost derivation

- Leftmost derivation digunakan dalam top-down parsing
- Rightmost derivation digunakan dalam bottom-up parsing

5.7 Parse Tree

Parse tree (juga disebut derivation tree atau concrete syntax tree) adalah representasi visual dari bagaimana string diturunkan dari grammar.

5.7.1 Struktur Parse Tree

Parse tree memiliki struktur berikut:

- **Root**: Labeled dengan start symbol S
- **Internal nodes**: Nonterminal symbols
- **Leaves**: Terminal symbols (dari kiri ke kanan membentuk input string)
- **Edges**: Menunjukkan aplikasi aturan produksi

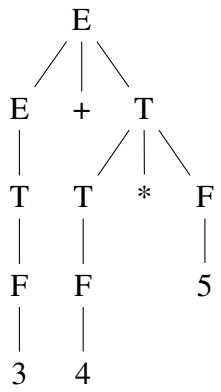
5.7.2 Contoh Parse Tree

Untuk ekspresi `3 + 4 * 5` dengan grammar sebelumnya, parse tree-nya adalah:

5.7.3 Parse Tree vs Abstract Syntax Tree (AST)

Perbedaan penting antara parse tree dan AST:

- **Parse Tree (Concrete Syntax Tree)**:
  - Mencakup semua detail sintaksis, termasuk punctuation
  - Setiap node sesuai dengan aturan produksi

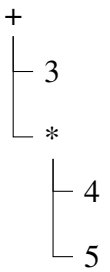


Gambar 5.6: Parse tree untuk ekspresi `3 + 4 * 5`

  - Lebih verbose, mencakup informasi yang tidak diperlukan untuk fase selanjutnya

- **Abstract Syntax Tree (AST)**:
  - Menghilangkan detail sintaksis yang tidak relevan (seperti parentheses grouping yang sudah jelas dari struktur)
  - Fokus pada struktur semantik program
  - Lebih kompak dan efisien untuk analisis semantik dan code generation

Contoh: Untuk ekspresi `3 + 4 * 5`, AST-nya lebih sederhana:



Gambar 5.7: AST untuk ekspresi `3 + 4 * 5`

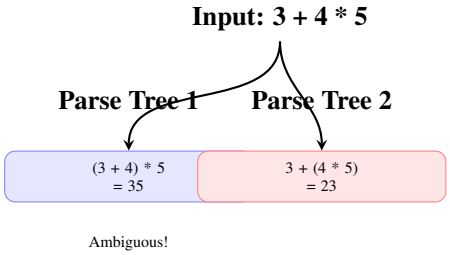
AST menghilangkan node-node intermediate seperti E, T, F yang tidak diperlukan untuk pemahaman semantik.

5.8 Ambiguity dalam Grammar

5.8.1 Definisi Ambiguity

Grammar dikatakan **ambiguous** jika terdapat setidaknya satu string dalam bahasa yang dapat memiliki lebih dari satu parse tree (atau derivation) yang berbeda. Ambiguity adalah masalah karena dapat menyebabkan interpretasi yang berbeda dari program yang sama.

Gambar 5.8 menunjukkan contoh ambiguity dalam grammar.



Gambar 5.8: Contoh ambiguity: dua parse tree berbeda untuk input yang sama

5.8.2 Contoh Grammar Ambiguous

Pertimbangkan grammar berikut untuk ekspresi:

```
E → E + E | E * E | number
```

Grammar ini ambiguous karena ekspresi `3 + 4 * 5` dapat di-parse dengan dua cara:
Parse Tree 1 (mengasumsikan + memiliki precedence lebih tinggi):

```
      E
     /|\
    E + E
   /|   |\
  E * E  5
  |   |
  3   4
```

Ini akan mengevaluasi sebagai `(3 + 4) * 5 = 35`
Parse Tree 2 (mengasumsikan * memiliki precedence lebih tinggi):

```
      E
     /|\
    E * E
   /|   |
  E + E 5
  |   |
  3   4
```

Ini akan mengevaluasi sebagai `3 + (4 * 5) = 23`

5.8.3 Mengatasi Ambiguity

Ada beberapa cara untuk mengatasi ambiguity:

1. **Menulis Grammar yang Unambiguous**: Menggunakan grammar yang secara eksplisit mendefinisikan precedence dan associativity. Contoh:

```
E → E + T | E − T | T
T → T * F | T / F | F
F → ( E ) | number
```

Grammar ini unambiguous karena:

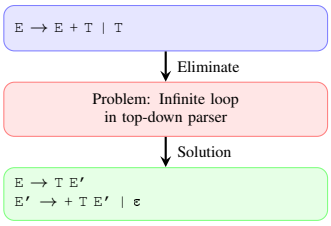
- Precedence: * dan / lebih tinggi dari + dan - (karena T lebih dalam dari E)
- Associativity: Left-associative untuk semua operator (karena left-recursive grammar)

2. **Disambiguating Rules**: Beberapa parser generator (seperti Yacc/Bison) memungkinkan penentuan precedence dan associativity secara eksplisit tanpa mengubah grammar.

3. **Operator Precedence Parsing**: Menggunakan tabel precedence untuk menentukan urutan evaluasi.

5.9 Left Recursion dan Left Factoring

Gambar 5.9 menunjukkan konsep left recursion.



Gambar 5.9: Left recursion dan eliminasi

5.9.1 Left Recursion

Left recursion terjadi ketika nonterminal muncul di posisi paling kiri dari produksinya sendiri. Contoh:

```
E → E + T | T
```

Left recursion dapat menyebabkan masalah pada top-down parser (khususnya recursive descent) karena dapat menyebabkan infinite loop. Parser akan terus mencoba mem-parse E tanpa pernah maju.

Eliminasi Left Recursion:
Grammar dengan left recursion:

```
A → A a | b
```

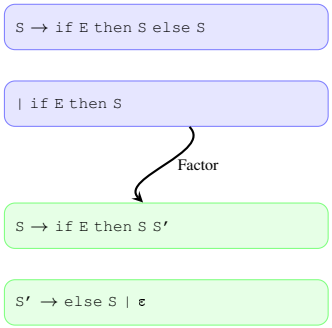
Dapat diubah menjadi:

```
A  → b A'
A' → a A' | epsilon
```

Contoh: `E → E + T | T` menjadi:

```
E  → T E'
E' → + T E' | epsilon
```

Gambar 5.10 menunjukkan konsep left factoring.



Gambar 5.10: Left factoring: sebelum dan sesudah

5.9.2 Left Factoring

Left factoring diperlukan ketika beberapa produksi dimulai dengan simbol yang sama, membuat parser tidak dapat memutuskan produksi mana yang harus digunakan tanpa lookahead lebih lanjut.

Contoh grammar yang membutuhkan left factoring:

```
S → if E then S else S
  | if E then S
```

Setelah left factoring:

```
S → if E then S S'
S' → else S | epsilon
```

5.10 Contoh Praktis: Grammar untuk Ekspresi Aritmatika

Mari kita buat grammar lengkap untuk ekspresi aritmatika yang dapat menangani:

- Operasi: +, -, *, /, mod
- Precedence: * dan / lebih tinggi dari + dan -

- Associativity: Left-associative
- Parentheses untuk grouping
- Unary minus
- Integer dan floating point numbers

Grammar dalam EBNF:

```
expression = term { ("+" | "-") term }
term       = factor { ("*" | "/" | "mod") factor }
factor     = [ "-" ] ( "(" expression ")" | number )
number     = integer | float
integer    = digit { digit }
float      = integer "." integer
digit      = "0" | "1" | ... | "9"
```

Atau dalam BNF:

```
<expression> ::= <term> | <expression> + <term> | <expression> - <term>
<term>       ::= <factor> | <term> * <factor> | <term> / <factor>
                 | <term> mod <factor>
<factor>     ::= - <factor> | ( <expression> ) | <number>
<number>     ::= <integer> | <float>
<integer>    ::= <digit> | <integer> <digit>
<float>      ::= <integer> . <integer>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

5.11 Manual Parsing: Latihan Derivation

Mari kita lakukan manual parsing untuk ekspresi `2 + 3 * 4` menggunakan grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

Leftmost Derivation:

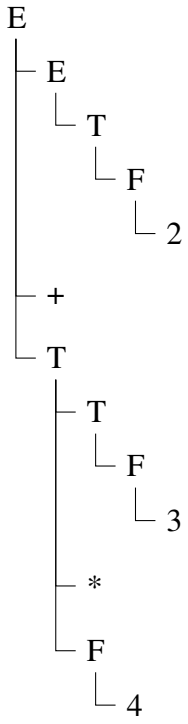
1. $E \Rightarrow E + T$
2. $E + T \Rightarrow T + T$
3. $T + T \Rightarrow F + T$
4. $F + T \Rightarrow 2 + T$
5. $2 + T \Rightarrow 2 + T * F$

6. $2 + T * F \Rightarrow 2 + F * F$

7. $2 + F * F \Rightarrow 2 + 3 * F$

8. $2 + 3 * F \Rightarrow 2 + 3 * 4$

Parse Tree:



Gambar 5.11: Parse tree untuk ekspresi `2 + 3 * 4`

5.12 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Context-free grammar adalah alat formal untuk mendefinisikan syntax bahasa pemrograman
2. BNF dan EBNF adalah notasi untuk menulis grammar
3. Derivation menunjukkan bagaimana string dihasilkan dari grammar
4. Parse tree merepresentasikan struktur sintaksis program
5. Ambiguity dalam grammar dapat menyebabkan interpretasi yang berbeda
6. Left recursion dan left factoring adalah isu penting dalam parsing

Pemahaman tentang CFG dan konsep-konsep terkait ini menjadi dasar penting untuk mempelajari teknik parsing (top-down dan bottom-up) yang akan dibahas dalam bab-bab selanjutnya.

5.13 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang context-free grammar dan parsing, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [2] - Bab 4: Syntax Analysis
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 3: Scanners dan Bab 4: Parsers
- **UC San Diego CSE 231**: Course materials tentang syntax analysis [4]
- **Northeastern University CS 4410**: Materials tentang parsing techniques [5]
- **Johns Hopkins University EN.601.428**: Course tentang syntax trees dan parsing [6]

Daftar Pustaka

[1] Diznr. *Six Phases of Compiler*. Online educational resource. 2024. URL: https://diznr.com/six-phases-of-compiler-lexical-syntax-semantic-intermediate-code-generation-optimization-code/.

[2] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.

[4] UC San Diego CSE Department. *CSE 231: Compiler Construction / Advanced Compiler Design*. Course materials and syllabus. 2024. URL: https://catalog.ucsd.edu/archive/2024-25/courses/CSE.html.

[5] Northeastern University. *CS 4410/6410: Compiler Design*. Course syllabus and materials, taught by Benjamin Lerner. 2024. URL: https://course.ccs.neu.edu/cs4410sp25/.

[6] Johns Hopkins University. *EN.601.428/628: Compilers and Interpreters*. Course syllabus and materials, taught by David Hovemeyer. 2024. URL: https://jhucompilers.github.io/fall2025/syllabus.html.