NIM: 241524033
Nama: Arkan Ramadhan Nugraha
Kelas: 2B-TI4
Mata Kuliah: Programming Language Pragmatics

Tugas 1 PPT

Pertanyaan 1

Kenapa if (x) valid tapi if x error menurut grammar?

Jawab:

Grammar CFG/BNF mendefinisikan bentuk if statement sebagai:

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

Aturan tersebut mengharuskan

1. Aturan ini mewajibkan:
2. Setelah keyword `if`, harus selalu ada tanda kurung buka `(`.
3. Di dalam tanda kurung harus ada sebuah ekspresi yang bisa dievaluasi menjadi *true* atau *false*.
4. Setelah tanda kurung tutup `)`, harus ada sebuah statement.

Maka, `if (x) {...}` valid karena sesuai grammar (`"if" "(" <expression> ")" <statement>`). `if x {...}` error karena tidak ada tanda kurung, sehingga tidak cocok dengan aturan grammar. Jadi, untuk yang valid itu pasti akan memenuhi, setelah “if” dicek dan cocok, lalu akan mengecek token/karakter selanjutnya, yang harus berupa “(“. Yang valid, ini akan cocok, karena benar dikasih “(“, dan lanjut ke <expresssion>, sedangkan `if x {…}` tidak valid, tidak cocok di pengecekkan “(“ yang dikasih sutau expression, bukan “(“

1. if (x) {...}
    1. “if” cocok
    2. “(“ cocok
    3. <expression>, x, cocok
    4. “)” cocok
    5. <statement>, cocok
2. if x {...}
    1. “if” cocok.

2. “(“ tidak cocok, dikasih identifier x instead of “(“, sehingga parsing gagal, dan compiler akan kasih error expected ‘(‘ after ‘if’

Jadi, if x itu error karena sintaks nya tidak sesuai dengan aturan grammar nya

Pertanyaan 2 dan 3

Grammar BNF table

Struktur	Grammar BNF	Contoh Valid	Contoh Error
While	`<while-statement> ::= "while" "(" <expression> ")" <statement>`	`while (x < 5) { x++; }`	`while x < 5 { x++; }`
For	`<for-statement> ::= "for" "(" <init> ";" <condition> ";" <increment> ")" <statement>`	`for (int i=0; i<10; i++) { printf("%d", i); }`	`for i=0; i<10; i++ { printf("%d", i); }`
Deklarasi Variable	`<declaration> ::= <type> <identifier> ";" <type> ::= "int" | "float" | "char"`	`int a; float b; char c;`	`int; float; char;`

Pertanyaan 4

CFG/BNF memberi aturan formal tentang bagaimana syntax harus ditulis. Dengan CFG/BNF, kita bisa tahu kenapa kode tertentu valid atau error: biasanya karena melanggar struktur grammar (misalnya tanda kurung hilang, identifier tidak ada, dll). Jadi, grammar membantu memahami logika compiler dalam membaca kode, serta membantu programmer menulis kode yang benar.

Tugas 2

BNF C

Grammar

```
<program> ::= <declaration_list>
```

```
<declaration_list> ::= <declaration> | <declaration>
<declaration_list>

<declaration> ::= <function_declaration> | <variable_declaration>

<function_declaration> ::= <type_specifier> <identifier> "("
<parameter_list> ")" <compound_statement>

<parameter_list> ::= <parameter> | <parameter> "," <parameter_list> |
ε

<parameter> ::= <type_specifier> <identifier>

<variable_declaration> ::= <type_specifier> <identifier> ";"

<type_specifier> ::= "int" | "float" | "char" | "void"

<compound_statement> ::= "{" <statement_list> "}"

<statement_list> ::= <statement> | <statement> <statement_list> | ε

<statement> ::= <expression_statement> | <return_statement>

<expression_statement> ::= <expression> ";"

<return_statement> ::= "return" <expression> ";"

<expression> ::= <assignment_expression>

<assignment_expression> ::= <identifier> | <identifier> "="
<arithmetic_expression>

<arithmetic_expression> ::= <term> | <arithmetic_expression> "+"
<term> | <arithmetic_expression> "-" <term>

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

<factor> ::= <identifier> | <number> | "(" <arithmetic_expression>
")"

<identifier> ::= <letter> | <identifier> <letter> | <identifier>
<digit>

<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"

<number> ::= <digit> | <number> <digit>
```

Contoh Program yang Digunakan

```
int add(int a, int b) {
```

```
    return a + b;

}
```

Derivation

<program>

⇒ <declaration_list>

⇒ <declaration>

⇒ <function_declaration>

⇒ <type_specifier> <identifier> "(" <parameter_list> ")" <compound_statement>

⇒ "int" <identifier> "(" <parameter_list> ")" <compound_statement>

⇒ "int" "add" "(" <parameter_list> ")" <compound_statement>

⇒ "int" "add" "(" <parameter> "," <parameter_list> ")" <compound_statement>

⇒ "int" "add" "(" <type_specifier> <identifier> "," <parameter_list> ")" <compound_statement>

⇒ "int" "add" "(" "int" <identifier> "," <parameter_list> ")" <compound_statement>

⇒ "int" "add" "(" "int" "a" "," <parameter_list> ")" <compound_statement>

⇒ "int" "add" "(" "int" "a" "," <parameter> ")" <compound_statement>

⇒ "int" "add" "(" "int" "a" "," <type_specifier> <identifier> ")" <compound_statement>

⇒ "int" "add" "(" "int" "a" "," "int" <identifier> ")" <compound_statement>

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" <compound_statement>

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" <statement_list> "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" <statement> "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" <return_statement> "}"

```
⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return"
<expression> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return"
<assignment_expression> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return"
<arithmetic_expression> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return"
<arithmetic_expression> "+" <term> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return" <term> "+"
<term> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return" <factor>
"+" <term> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return"
<identifier> "+" <term> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return" "a" "+"
<term> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return" "a" "+"
<factor> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return" "a" "+"
<identifier> ";" "}"

⇒ "int" "add" "(" "int" "a" "," "int" "b" ")" "{" "return" "a" "+"
"b" ";" "}"
```

Parse Tree

<program>

|

<declaration_list>

|

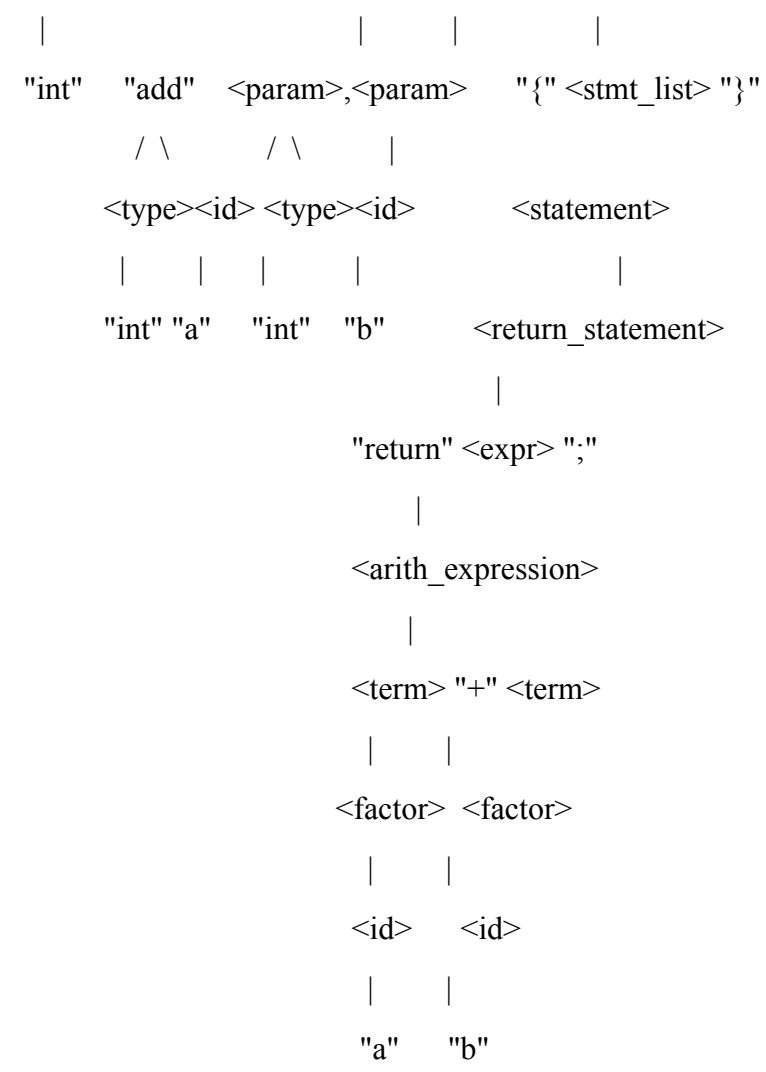
<declaration>

|

<function_declaration>

/ | | | | \

<type_specifier> <id> "(" <param_list> ")" <compound_statement>



AST

Program

└── FunctionDeclaration

  ├── ReturnType: int

  ├── Name: add

  ├── Parameters

  │ ├── Parameter(type: int, name: a)

  │ └── Parameter(type: int, name: b)

  └── Body

    └── ReturnStatement

      └── BinaryExpression

├── Operator: +

├── Left: Identifier(a)

└── Right: Identifier(b)

Dokumentasi Proses Pembentukan Grammar dan Alasan Desain

Sumber dan Metodologi Grammar:

- Referensi: ISO C11 Standard (ISO/IEC 9899:2011) dan GCC parser implementation
- Alasan Penyederhanaan: Grammar C penuh (800+ rules) terlalu kompleks untuk derivation manual
- Focus Area: Function definition, variable declaration, expressions, statements
- Kompleksitas yang Dihilangkan: Preprocessor, advanced types, control flow, storage classes

Konversi ISO Standard -> BNF:

- Precedence Preservation: Tetap mengikuti C standard (15 levels)
- Left Recursion Elimination: Dikonversi untuk top-down parsing
- Ambiguity Resolution: Dangling else dihindari dengan subset selection
- Context Sensitivity: Type checking diasumsikan ditangani semantic analyzer

Token Dependencies:

- Lexical Analysis: Lexer memproduksi IDENTIFIER, NUMBER, KEYWORD tokens
- Whitespace Handling: Lexer menghilangkan whitespace dan comments
- Grammar Assumptions: Parser menerima token stream, bukan character stream

Grammar BNF Lengkap:

- 18 Non-terminals dengan hierarki yang jelas
- 25 Terminals termasuk operators dan keywords
- 28 Production Rules total
- Precedence Table sesuai C Standard dengan 15 levels

Tokenisasi Detail:

- 5 Kategori Token: Keywords, Identifiers, Literals, Operators, Delimiters
- Lexer Responsibilities: Whitespace removal, comment handling, token classification
- Multi-character Operators: `<=`, `>=`, `==`, `!=`, `&&`, `||`

EBNF Python

Grammar

```
Terminals: NAME, NUMBER, STRING, NEWLINE, INDENT, DEDENT, EOF Symbols (literal): "def" ":" "(" ")" "," "=" "+" "-" "*" "/" "//" "%" "**" "return"

program ::= { statement } EOF

statement ::= simple_stmt | compound_stmt

simple_stmt ::= (assign_stmt | return_stmt | expr_stmt) NEWLINE

compound_stmt ::= funcdef

funcdef ::= "def" NAME "(" [ param_list ] ")" ":" NEWLINE INDENT { statement } DEDENT

param_list ::= param { "," param } param ::= NAME

assign_stmt ::= target "=" expr target ::= NAME

expr_stmt ::= expr

return_stmt ::= "return" [ expr ]

expr ::= disjunction

disjunction ::= conjunction { "or" conjunction } conjunction ::= inversion { "and" inversion } inversion ::= "not" inversion | comparison

comparison ::= arith { comp_op arith } comp_op ::= "==" | "!=" | "<" | ">" | "<=" | ">=" | "in" | "is"
```

```
arith ::= term { ("+" | "-") term } term ::= factor { ("*" | "/" |
"//" | "%") factor } factor ::= ("+" | "-" | "~") factor | power
power ::= atom [ "**" factor ]

atom ::= NAME | NUMBER | STRING | "(" expr ")" | "[" [ expr_list ]
"]"

expr_list ::= expr { "," expr } [ "," ]
```

Contoh Program yang Akan Digunakan

```
def add(a, b):
  c = a + b
  return c
```

Derivation

1. `program` → { `statement` } `EOF` → satu `statement` + `EOF`
2. `statement` → `compound_stmt`
3. `compound_stmt` → `funcdef`
4. `funcdef` → `"def" NAME "(" [ param_list ] ")" ":" NEWLINE INDENT { statement } DEDENT`
5. `NAME` → `add`
   param_list → `param { "," param }` → `param "," param` → `a , b`
6. Sekarang isi blok: `{ statement }` → dua `statement` (assignment dan return)
7. statement1 → `simple_stmt` → `assign_stmt NEWLINE`
8. `assign_stmt` → `target "=" expr`
9. `target` → `c`
10. `expr` → `arith` → `term` → `factor` → `power` → `atom` → `NAME` → `a`
11. after `+` term → `b` → jadi `expr = a + b`
12. statement2 → `simple_stmt` → `return_stmt NEWLINE`

13. `return_stmt → return expr → expr → NAME → c`

14. Tutup `DEDENT`, `EOF`. → string lengkap sama dengan program.

Parse Tree

```
program
└─ statement (compound_stmt)
  └─ funcdef
    ├─ "def"
    ├─ NAME -> "add"
    ├─ "("
    ├─ param_list
    │  ├─ param -> NAME -> "a"
    │  ├─ ","
    │  └─ param -> NAME -> "b"
    ├─ ")"
    ├─ ":"
    ├─ NEWLINE
    ├─ INDENT
    ├─ statement (simple_stmt)
    │  └─ assign_stmt
    │    ├─ target -> NAME -> "c"
    │    ├─ "="
    │    └─ expr
    │      └─ arith
    │        ├─ term
    │        │  └─ factor -> power -> atom -> NAME -> "a"
    │        ├─ "+"
    │        └─ term
```

```
│        └─ factor -> power -> atom -> NAME -> "b"
│  └─ NEWLINE
├─ statement (simple_stmt)
│  └─ return_stmt
│    ├─ "return"
│    └─ expr -> NAME -> "c"
│  └─ NEWLINE
└─ DEDENT
```

AST

```
Module
└─ FunctionDef (name="add", params=["a", "b"])
  ├─ Assign
  │  ├─ Target: Name("c")
  │  └─ Value: BinOp(+)
  │    ├─ Left: Name("a")
  │    └─ Right: Name("b")
  └─ Return
    └─ Value: Name("c")
```

Dokumentasi Proses Pembentukan Grammar dan Alasan Desain

1. Sumber: PEG grammar CPython dari referensi digunakan sebagai referensi utama (mis. `function_def`, `params`, `suite`, `assign_stmt`, `return_stmt`, serta aturan ekspresi dan prioritas operator).
2. Mengapa disederhanakan?
    1. Supaya bisa dijabarkan manual (derivation & parse tree). grammar CPython penuh terlalu besar.
    2. Fokus: `def`-function, parameter list, assignment, return, ekspresi aritmetika. Fitur lain (comprehensions, decorators, imports, try/with, pattern matching) dihilangkan untuk menghindari kompleksitas yang tidak diperlukan.

3. Konversi PEG → EBNF:
    1. Alternatif PEG yang menggunakan | (ordered choice) saya ubah ke alternatif EBNF (masih terpadu).
    2. Lookaheads (& / !) di PEG (digunakan di grammar CPython untuk disambiguasi) *tidak* saya masukkan karena subset ini sederhana; seandainya diperlukan (mis. untuk membedakan `NAME "("` vs `NAME` dalam target), parser implementasi harus menambahkan lookahead di lexer/parser.
4. Indentation:
    1. CPython PEG menghasilkan token `INDENT`/`DEDENT`/`NEWLINE`. Grammar EBNF ini juga mengandalkan token-token tersebut; lexer harus tetap memproduksinya. Tanpa token indent/dedent, blok tidak bisa dikenali.
5. Precedence operator: disusun sedemikian agar urutan evaluasi sesuai Python (power → factor → term → arith → compare → and/or). Ini mengikuti struktur di PEG aslinya (mis. `sum`, `term`, `factor`, `power` di PEG).
6. Tokenisasi tanggung jawab lexer: rules ini mengasumsikan lexer memisahkan `NAME`, `NUMBER`, `STRING`, serta menyediakan NEWLINE/INDENT/DEDENT. Grammar tidak menangani whitespace/komentar.
7. Ambiguitas & Praktik: grammar EBNF ini deterministic untuk contoh yang diberikan. Pada implementasi nyata (parser generator) mungkin perlu memakai PEG (no backtracking or explicit lookahead) atau LL/LR parser dengan penanganan left-recursion (atau transformasi) sesuai alat.

Referensi

https://docs.python.org/3/reference/grammar.html

ISO C11 Standard (ISO/IEC 9899:2011) dan GCC parser implementation

https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm