Bab 6

Semantic Analysis dan Error Handling

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 3.3:** Menangani semantic error dengan pesan yang informatif

6.1 Abstract Syntax Tree (AST) Deep Dive

Abstract Syntax Tree (AST) adalah representasi internal program yang telah disederhanakan. Analisis semantik bertugas memastikan bahwa program memiliki makna yang valid sesuai dengan aturan bahasa pemrogramannya, di luar sekadar kebenaran struktur sintaksisnya [1].

6.1.1 Struktur Node AST

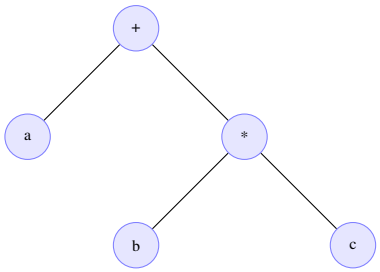
Setiap node dalam AST mewakili konstruk bahasa (misal: `IfStmt`, `BinaryExpr`).

1. **Expression Nodes**: Literal, Identifier, Unary/Binary operations.
2. **Statement Nodes**: Assignment, Loop, Conditional branches.
3. **Declaration Nodes**: Variabel, Fungsi, Tipe.

6.2 Sistem Tipe dan Type Checking

6.2.1 Tipe Data dalam Kompilator

Sistem tipe mendefinisikan aturan bagaimana tipe data diasosiasikan dengan ekspresi. Tipe dasar meliputi `int`, `float`, `bool`, dan `void`.



Gambar 6.1: Representasi AST untuk ekspresi `a + b * c`

6.2.2 Algoritma Type Checking

Type checking dilakukan dengan menelusuri (*traversing*) AST secara *post-order*.

1. Untuk leaf node: ambil tipe dari symbol table atau literal.

2. Untuk node internal: verifikasi kompatibilitas tipe operan (misal: `int + float` mungkin memerlukan promosi tipe).

```
Type checkBinaryExpr(BinaryExpr* node) {
    Type left = check(node->left);
    Type right = check(node->right);
    if (!isCompatible(left, right)) {
        error("Type mismatch: " + left.str() + " and " + right.str());
    }
    return resultType(left, right);
}
```

6.3 Analisis Kontekstual dan Validasi

Selain tipe, kompilator juga melakukan pengecekan aturan kontekstual:

- **Control Flow**: Memastikan statement `break` atau `continue` hanya muncul di dalam loop.

- **Function Signature**: Verifikasi jumlah dan tipe parameter pada pemanggilan fungsi.

- **Return Type**: Memastikan statement `return` memberikan nilai sesuai tanda tangan fungsi.

6.3.1 Error Reporting yang Informatif

Setiap kesalahan harus dilaporkan dengan posisi (baris dan kolom) serta konteks yang jelas untuk membantu pemrogram memperbaiki kode.

6.4 Praktikum: Implementasi Type Checker

Mahasiswa akan mengimplementasikan *semantic analyzer* sederhana menggunakan pola *Visitor Pattern* untuk menelusuri AST.

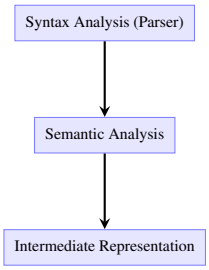
6.4.1 Visitor Pattern

Pola ini memisahkan algoritma analisis dari struktur data AST, sehingga memudahkan penambahan aturan semantik baru tanpa mengubah kelas node AST.

```
class TypeChecker : public ASTVisitor {
    void visit(BinaryExpr* node) override {
        // Cek tipe operan biner
    }
    void visit(Assignment* node) override {
        // Cek kecocokan tipe variabel dan nilai
    }
};
```

6.5 Kesimpulan Analisis Semantik

Analisis semantik menjembatani antara tugas parser (struktur) dan code generator (instruksi). Keberhasilan fase ini menjamin bahwa program yang dikompilasi memiliki makna yang konsisten dan mematuhi spesifikasi bahasa.



Gambar 6.2: Aliran informasi melalui fase semantik

Aktivitas Pembelajaran

1. **Semantic Rules**: Definisikan semantic rules untuk bahasa sederhana.
2. **AST Builder**: Implementasikan AST construction dari parse tree.
3. **Type Checker**: Bangun type checker dengan error reporting informatif.
4. **Error Recovery**: Implementasikan error recovery untuk semantic errors.

5. **Symbol Table Integration**: Integrasikan semantic analyzer dengan symbol table.

Latihan dan Refleksi

1. Identifikasi semantic errors dalam potongan kode yang diberikan!
2. Buat semantic rules untuk function calls dan parameter passing!
3. Implementasikan type checker untuk expressions dengan multiple types!
4. Desain error messages yang informatif untuk berbagai semantic errors!
5. Analisis trade-off antara strict vs lenient semantic checking!
6. **Refleksi**: Bagaimana semantic analysis mempengaruhi kualitas compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 3.3

A. Pilihan Ganda

1. Semantic analyzer bertugas untuk:
   (a) Memverifikasi syntax correctness
   (b) Memverifikasi semantic correctness
   (c) Mengoptimasi kode
   (d) Mengenerate kode target
2. Error reporting yang baik harus mencakup:
   (a) Error message saja
   (b) Line number saja
   (c) Line number, column, dan context
   (d) Hanya error code
3. Type coercion adalah:
   (a) Error handling mechanism
   (b) Automatic type conversion

(c) Type checking algorithm

(d) Optimization technique

B. Essay

1. Jelaskan strategi error recovery dalam semantic analysis dan berikan contoh!
2. Desain dan implementasikan semantic analyzer untuk bahasa dengan variabel assignment dan arithmetic expressions!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat menangani semantic error dengan pesan yang informatif

☐ Saya dapat mengimplementasikan syntax-directed translation

☐ Saya dapat membangun AST dari parse tree

☐ Saya dapat mengintegrasikan semantic analyzer dengan symbol table

☐ Saya dapat mendesain error recovery strategies

☐ Saya dapat mengimplementasikan type checking dengan coercion

Rangkuman

Bab ini membahas semantic analysis dan error handling, termasuk syntax-directed translation, type system implementation, error detection, dan recovery strategies. Mahasiswa belajar membangun semantic analyzer yang robust.

Poin Kunci:

- Semantic analysis memverifikasi meaning dan correctness program
- Syntax-directed translation menggabungkan parsing dengan semantic analysis
- Type checking memastikan type compatibility dan consistency
- Error reporting yang baik memberikan informasi yang jelas dan berguna
- Error recovery memungkinkan compiler melanjutkan proses compilation

Kata Kunci: ***Semantic Analysis***, ***Syntax-Directed Translation***, ***Type Checking***, ***Error Handling***, ***AST***, ***Type Coercion***, ***Error Recovery***

Daftar Pustaka

[1] Nguyen Thanh Vu. *Compiler Class Notes: Semantic Analysis*. Class notes. 2024. URL: https://nguyenthanhvuh.github.io/class-compilers/notes/sem.html.