BAB 6

KESIMPULAN DAN SARAN

6.1 Kesimpulan

Perangkat lunak yang dapat melakukan *parsing* kode sumber Javascript ke dalam struktur data *abstract syntax tree* dan *control flow graph* telah berhasil dibangun. *Parsing* dilakukan dengan dua proses yang diadopsi dari *compiler*, yaitu proses analisis leksikal dan proses analisis sintaksis. Dua proses tersebut menghasilkan AST yang selanjutnya digunakan untuk membangun CFG. Kemudian, AST divisualisasikan menggunakan notasi JSON dan gambar visualisasi struktur data pohon, dan CFG divisualisasikan menggunakan gambar graf berarah. Dilakukan pengujian untuk mengukur kualitas dari perangkat lunak, yaitu pengujian terhadap fungsionalitas dan juga performa dari perangkat lunak.

Secara umum, perangkat lunak berfungsi dengan baik terbukti dengan hasil pengujian fungsional yang sudah dilakukan dan dilaporkan pada Subbab 5.2. Akan tetapi terdapat satu kasus uji yang menghasilkan CFG yang tidak sesuai harapan. Kode program pada kasus uji tersebut memiliki *statement* yang berada di dalam sebuah *expression* menghasilkan CFG yang tidak sempurna; *statement* yang berada di dalam *expression* tidak ada dan tidak direpresentasikan di dalam CFG.

Telah dilakukan juga pengujian eksperimental untuk menguji performa dari perangkat lunak dalam melakukan *parsing* dan visualisasi. Perangkat lunak mampu melakukan *parsing* 200 baris kode sumber dengan waktu rata-rata 2053,8 milidetik dan kode sumber dengan nilai kompleksitas sintaks 200 dengan waktu rata-rata 1734,7 milidetik. Hasil eksperimen dianalisis dan dapat disimpulkan bahwa semakin banyak baris kode program dan semakin kompleks kode program, maka semakin lama waktu yang dibutuhkan untuk melakukan *parsing*.

6.2 Saran

Saran untuk penelitian selanjutnya adalah sebagai berikut:

- Melakukan pengujian lebih lanjut untuk menyempurnakan pembangunan AST dan CFG.
  Pengujian yang dilakukan memiliki lingkup yang terbatas. Ada banyak kasus pemakaian sintaks kompleks, yaitu ketika beberapa *expression*, *statement*, atau *pattern* dipakai secara bersamaan dalam satu *statement*, belum diuji kebenaran hasil AST dan CFG-nya.

- Membuat visualisasi yang lebih spesifik untuk memvisualisasikan AST dan CFG.
  Visualisasi AST dan CFG dilakukan menggunakan *library* yang dibuat dan ditujukan untuk pemakaian umum, sehingga hasil gambar hasil visualisasi AST dan CFG kurang maksimal. Sebagai contoh, CFG yang dihasilkan hanya menampilkan nama node sintaks tanpa adanya perincian dari kode program yang direpresentasikan karena keterbatasan *format library*.

- Menangani dan menambahkan pesan *error*.
  Kode program masukan yang tidak sesuai dengan spesifikasi perangkat lunak (Javascript ES6) tidak bisa di-*parse* oleh perangkat lunak. Untuk menanganinya, pesan *error* bisa ditambahkan untuk memberitahu pengguna kesalahan yang terdapat pada kode program sesuai dengan spesifikasi Javascript ES6.

- Menambahkan spesifikasi ECMAScript selanjutnya.
  Perangkat lunak saat ini mampu melakukan *parsing* terhadap kode program Javascript sesuai dengan standar spesifikasi ECMAScript versi 6 tahun 2015, atau ECMAScript2015. Dokumen spesifikasi tersebut, ECMA-262, diperbaharui setiap tahunnya. Pada pertengahan tahun 2020, *draft* dari spesifikasi ECMAScript2021 sudah diterbitkan.

- Membangun kode program dari AST dan/atau CFG.
  AST dan CFG merepresentasikan struktur dari suatu kode program. Maka, dari struktur kode program yang ada pada AST dan CFG, kode sumber program, dalam berbagai bahasa pemrograman, dapat dibangun kembali.

- Melakukan parsing CFG dari kode program
  Pada penelitian ini, CFG dibangun dengan membuat AST terlebih dahulu, lalu menelusuri AST tersebut untuk membangun CFG. Akan tetapi, CFG dapat langsung dibangun dengan menganalisis kode program. CFG yang dibangun juga dapat memiliki lingkup yang lebih luas dan lebih lengkap.

DAFTAR REFERENSI

[1] Bondy, J. A. dan Murty, U. S. R. (1976) *Graph Theory with Applications*. Elsevier, New York.

[2] Aho, A. V., Lam, M. S., Sethi, R., dan Ullman, J. D. (2006) *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Overbey, J. dan Johnson, R. (2008) Generating rewritable abstract syntax trees. *Software Language Engineering, First International Conference, SLE 2008, September 29-30, 2008*, Toulouse, France, 09, pp. 114–133. University of Illinois at Urbana-Champaign.

[4] Gold, R. (2010) Control flow graphs and code coverage. *Applied Mathematics and Computer Science*, **20**, 739–749.

[5] Watson, A. H., McCabe, T. J., dan Wallace, D. R. (1996) Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publications 500-235*, Gaithersburg, 9.

[6] Rosen, K. H. (2006) *Discrete Mathematics and Its Applications: And Its Applications*. McGraw-Hill Higher Education.

[7] McCabe, T. J. (1976) A complexity measure. *IEEE Trans. Softw. Eng.*, **2**, 308–320.

[8] Goodman, D., Morrison, M., Novitski, P., dan Rayl, T. G. (2010) *JavaScript Bible*, 7th edition. Wiley Publishing.

[9] Ecma International (2017) *The JSON Data Interchange Syntax*, 2nd edition. ECMA-International, Geneva.

[10] Ecma International (2015) *ECMAScript 2015 Language Specification*, 6th edition. ECMA-International, Geneva.

[11] Bergin, T. (2007) A history of the history of programming languages. *Commun. ACM*, **50**, 69–74.

[12] Chomsky, N. (1957) *Syntactic Structures*. Mouton and Co., The Hague.