課題 1 : XC-small 再帰下降型構文解析器

22B30862　情報工学系　山口友輝

2024 年 7 月 12 日

1　はじめに

私は今回、XC-small の再帰下降型構文解析器を実装するために、XC-small が LL(1) 文法でないことの証明と、xcc-small.c を修正することで parse_translation_unit 関数と unparse_AST 関数を実装した。

2　LL(1) でないことの証明

2.1　ラベル式と式文の衝突

$$A = Director(stetement,\ IDENTIFIER\ ":")$$

$$B = Director(stetement,\ [exp]\ ";")$$

と置いて、$A \cap B \neq \phi$ を示すことで、LL(1) ではないことを示す。このとき、

$$A = first(IDENTIFIER\ ":")$$

$$A = \{IDENTIFIER\}$$

また、

$$B = first([exp]\ ";")$$

$$B = \{INTEGER,\ CHARCTER,\ STRING,\ IDENTIFIER,\ "(",\ ";"\}$$

よって、

$$A \cap B = \{IDENTIFIER\} \neq \phi$$

となるので、この部分は LL(1) ではない。

2.2　ぶらぶら else の衝突

たとえば if(A) if(B) C else D のような文があったときに if 文の中に if-else 文が入っているのか、if-else 文の中に if 文が入っているのかが判別できない。このように、statement の if 文または if-else 文で書かれた文は構文木が一意に定まらないことがあるので、LL(1) ではない。

3　xcc-small.c の修正

3.1　parse_translation_unit 関数

parse_translation_unit 関数を実装するにあたり、他の非終端記号の parse の関数（parse_statement など）も実装した。

それぞれの関数を実装する上での基本的な方針としては、ast = create_AST(”xxx”, 0); によって、非終端記号のノードを作成し、子ノードの AST を ast に add_AST 関数によって付け足していき、構文木を構築した。子ノードの作成方法としては、xcc-small の文法を参照し、それが終端記号なら consume_token 関数で構文エラーを確認、トークンのポインタを 1 つ進めてから、子ノードを作成。非終端記号なら、その非終端記号に合った関数 (parse_*) によって子ノードを作成した。また、非終端記号のときの構文エラーの確認は、parse_*が動いたときに現在のポインタのトークンが動かした関数に対応する非終端記号の first 集合と一致するかを確認することで、構文エラーを処理した。また、非終端記号をどのように展開するかは、lookahead(1) で現在のポインタに対応するトークンを受け取り、switch 文によって判断した。

3.2　unparse_AST 関数

この関数の実装方針としては、第一引数 ast の ast_type を参照することで、どの非終端記号を処理するかを決定した。

それぞれの非終端記号を処理する際の基本方針としては、まず、printf_ns(depth, ””); によって、それぞれのネストの深さに応じたインデントを出力し、徐々に展開していったときに終端記号が期待されるトークンであったら出力、そうでなかったら、unparse_error 関数でプログラムを終了する。展開していったときに、非終端記号だったら、unparse_AST 関数を再帰敵に呼び出した。また、可読性向上のため、構文エラーの確認とトークンの出力を行う関数である、printf_token(struct AST *ast, char *type) を用いた。この関数は ast-¿ast_type と type が等しいときに type を出力、そうでないときに unparse_error 関数でプログラムを終了するという役割を果たしている。これによって、整数、文字、文字列、識別子以外の終端記号の構文エラー確認と出力を簡潔に行なった。これ以外の終端記号や、非終端記号の処理では構文エラーを検出し次第、unparse_error 関数を呼び出すようにした。

if 文と if-else 文の内部の statement{} をつける方法としては、内部の statement を展開したものが compound_statement の場合、{} が重複してしまうので、このときは {} をつけず、それ以外のときは {} をつけて出力するようにした。

4　考察

再帰下降型構文解析器は、機械的な文法に対しては非常に直感的かつ実装が容易である。今回の XC-small の実装としては、構文木を構築することによる構文解析を行なったが、これにより、構文エラー検出を容易に行うことができ、エラー箇所を構文エラーのメッセージから確認することができる。さらに、再帰的な関数呼び出しにより、各構文要素を分割して処理できるため、デバッグが容易である。例えば、構文木の各ノードを追跡することで、どの部分でエラーが発生しているのかを迅速に特定できる。これにより、問題のある箇所を効率的に修正することができ、開発のスピードと品質が向上する。

5　感想

今回の実装を通じて、再帰下降型構文解析器の基本的な仕組みを深く理解することがでた。特に、XC-small が LL(1) 文法ではないために生じる複雑さに対処する際に、多くの学びがあった。