言語処理系分科会
第 2 回 - AST

semiexp

今回の内容

- yacc で C++ を使うための方法
- AST の構成法
- AST の実行

yacc with C++

- 前回言ったように，lex は C しか使えないけど yacc は C++ が使える
- C だけで書くのは面倒…
- yacc に C++ コードを出力させる方法を説明します
- C だけで書くんだ，という人はここは無視してかまいません

yacc の出力を C++ にする

- 実は簡単で，yacc -o calc.cpp calc.y みたいにするだけ
  - -o [filename] とすると出力ファイル名が指定できる
- このとき，ヘッダファイルは y.tab.h ではなく calc.hpp に出力される
  - calc.l の #include も変えないといけない

C++ 側で C 関数を参照

- いろいろなものを extern “C” する必要がある
  - C コードで定義されるもの (lex 部分)
- int yylex(void); は extern “C” しないとだめです
  - パーサの入力を FILE * でなく const char * にしたい気分になると，もっといろいろなものを extern “C” しないといけなくなる
- 他のもの int yyparse(); などは extern “C” しなくてよい

- AST を作るあたりから，コードを分割したい欲求が出てきます
- ここらへんのものはヘッダファイルにまとめてしまいましょう

コンパイル手順

- lex の出力は C コードなので，そのまま g++ とかにかけると怒られる
- ここだけ gcc に任せる
  - gcc -c lex.yy.c -o lex.yy.o
    g++ lex.yy.o main.cpp parser.cpp -o calc.exe
    みたいな感じ
- これで，yacc で C++ を使えるようになった

AST の例

- たとえば Clang
- Expr (式), Stmt (文), Decl (宣言), Type (型) といった要素がある
- Expr, Stmt, Decl にはいろいろな種類があり，たとえば Expr では
  - BinaryOperator (二項演算)
  - DeclRefExpr (変数，関数宣言などの参照)
  - CallExpr (関数呼び出し)
  - IntegerLiteral (整数リテラル)
  - … などがある

AST の例

- Expr は式であるから，「値を持つ」という共通した特徴がある
- 例えば二項演算で，(値) [演算子] (値) といった構造を表現したいときに，値として「二項演算をとるもの」「整数リテラルをとるもの」…といちいち定義するのは非効率
- だから，二項演算は (Expr) [演算子] (Expr) と表す

AST をプログラム上で表す

- (OCaml を使うととても簡潔に表せますが…)
- ここでは C, C++ の方法を説明します

- 基本は，「1 つの構造体が AST 上の 1 つのノードを表す」
- 他の AST ノードを指したい時 (例えば二項演算の左右の値) はポインタで指す

C による方法

- Expr なら，どんな Expr かによらず全部 1 種類の構造体で表す
- それだけだと Expr の多様性を持たせられないので，Expr の種類ごとに特異な部分は union でまとめる
- また，その Expr がどの種類の Expr かを enum などで持たせる
- struct Expr {
  union {
  struct BinaryOperator bin;
  …
  };
  /* 共通な部分 (Expr の種類，ソース上の位置情報など) */
  int expr_kind;
  };

C++ による方法

- C による方法はあまりスマートではない
- C++ では，クラスの継承を用いるともう少しまともに書ける

- Expr クラスには，共通で持つべき情報を持たせる
- 各種 Expr を表すクラスは，Expr クラスを継承して作る
- すると，「式の評価」関数などは仮想関数を使って書ける

yacc で AST を得る

- パーサの各アクションで，適切に AST のノードを生成するだけ
- 例えば，
  | expr ADD expr
  {
  $$ = new BinOpExpr(BINOP_ADD, $1, $3);
  }
  みたいに（気分）書く
- 本当は，動的にノードを生成するところでいちいち new するのはあまりよくなさそうだけど…
- %union を AST のノードを保持できるようにしないといけないことに注意
  - そのままやると lex のコンパイルで困る
  - 解決策としては，ポインタを void * で保持する

AST の実行

- C++ なら，仮想関数を使って Expr, Stmt などに evaluate 関数を用意できる
- 「電卓」の範囲だったら，evaluate は本当にやるだけ
  - 変数，関数などが存在しない
- 普通は，環境 (environment) を使って変数や関数などを管理する

練習問題

1. yacc の出力を C++ にしてみよう (C で書く人はこの問題は無視してください)
2. 前回の「電卓」の文法構造に対する AST を設計しよう
   - できれば，プログラミング言語の AST に拡張しやすいように
3. 「電卓」のパーサを，計算しきった結果ではなく AST を返すようにしよう
4. その返された AST を受け取って，計算結果を返す関数を書こう
5. (参考) この AST に対して，pretty printer を作成しよう
   - AST に対する pretty printer は，AST を受け取って，その AST の構文的意味を読みやすい形にして（たとえば，ソースコード状にする）出力するもの
   - これがあると，AST の内容を確認できるようになってデバッグがしやすくなる