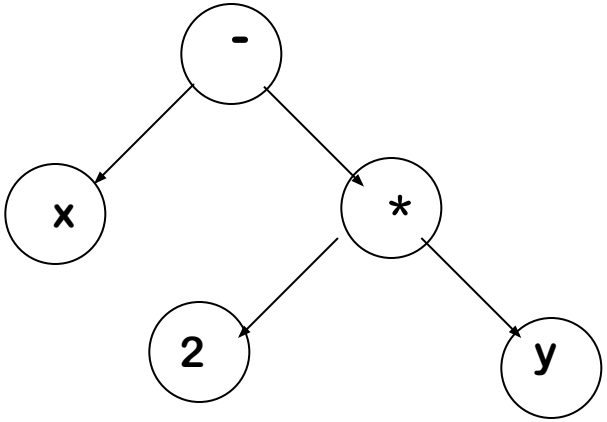
CENG513 Compiler Design and Construction
A Short Kaleidoscope Tutorial

Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

- Can use linearized form of the tree
  - Easier to manipulate than pointers

    `x 2 y * -` in postfix form

    `- * 2 y x` in prefix form

Kaleidoscope Language

- top → definition | external | expression | ;
- definition →def prototype expression
- prototype → id (arg)
- arg →arg | id | ε
- expression → primary binoprhs
- primary → identifierexpr | numberexpr | parenexpr
- identifierexpr → identifier | identifier (expression')
- expression' → expression' | expression | ε
- numberexpr →number
- parenexpr →(expression)
- binoprhs → + primary | ε

Kaleidoscope AST

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() = default;
 };


/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;

public:
  NumberExprAST(double Val) : Val(Val) {}
 };


/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;

public:
  VariableExprAST(const std::string &Name) : Name(Name) {}
};
```

Kaleidoscope AST

```
/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  std::unique_ptr<ExprAST> LHS, RHS;

public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
    : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<std::unique_ptr<ExprAST>> Args;

public:
  CallExprAST(const std::string &Callee,
              std::vector<std::unique_ptr<ExprAST>> Args)
    : Callee(Callee), Args(std::move(Args)) {}
};
```

Kaleidoscope AST

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;

public:
  PrototypeAST(const std::string &Name, std::vector<std::string> Args)
    : Name(Name), Args(std::move(Args)) {}

  const std::string &getName() const { return Name; }
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto;
  std::unique_ptr<ExprAST> Body;

public:
  FunctionAST(std::unique_ptr<PrototypeAST> Proto,
              std::unique_ptr<ExprAST> Body)
    : Proto(std::move(Proto)), Body(std::move(Body)) {}
 };
```

Basic Expression Parsing - primary → identifierexpr | numberexpr | parenexpr

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr();
  case tok_number:
    return ParseNumberExpr();
  case '(':
    return ParseParenExpr();
  }
}
```

Basic Expression Parsing - primary → identifierexpr | numberexpr | parenexpr

```
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);
  getNextToken(); // consume the number
  return std::move(Result);
 }


/// parenexpr ::= '(' expression ')'
static std::unique_ptr<ExprAST> ParseParenExpr() {
  getNextToken(); // eat (.
  auto V = ParseExpression();
  if (!V)
    return nullptr;

  if (CurTok != ')')
    return LogError("expected ')'");
  getNextToken(); // eat ).
  return V;
 }
```

Basic Expression Parsing - primary → identifierexpr | numberexpr | parenexpr

```
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return std::make_unique<VariableExprAST>(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<std::unique_ptr<ExprAST>> Args;
  if (CurTok != ')') {
    while (true) {
      if (auto Arg = ParseExpression())
        Args.push_back(std::move(Arg));
      else
        return nullptr;

      if (CurTok == ')')
        break;

      if (CurTok != ',')
        return LogError("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return std::make_unique<CallExprAST>(IdName, std::move(Args));
}
```

Driver

```
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (true) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:
      return;
    case ';': // ignore top-level semicolons.
      getNextToken();
      break;
    case tok_def:
      HandleDefinition();
      break;
    case tok_extern:
      HandleExtern();
      break;
    default:
      HandleTopLevelExpression();
      break;
    }
  }
}
```

Sample Execution

```
$ ./a.out
ready> def foo(x y) x+foo(y, 4.0);
Parsed a function definition.
ready> def foo(x y) x+y y;
Parsed a function definition.
Parsed a top-level expr
ready> def foo(x y) x+y );
Parsed a function definition.
Error: unknown token when expecting an expression
ready> extern sin(a);
ready> Parsed an extern
ready> ^D
 $
```

Kaleidoscope Language - Extended Repeat Until

- top → definition | external | expression | ;
- definition →def prototype expression
- prototype → id (arg)
- arg →arg | id | ε
- expression → primary binoprhs
- primary → identifierexpr | numberexpr | parenexpr | repeatexpr
- identifierexpr → identifier | identifier (expression')
- expression' → expression' | expression | ε
- numberexpr →number
- parenexpr →(expression)
- binoprhs → + primary | ε
- repeatexpr →repeat expression until expression

References

Kaleidoscope Lexer and Parser

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html