

Code Generation for Data Processing

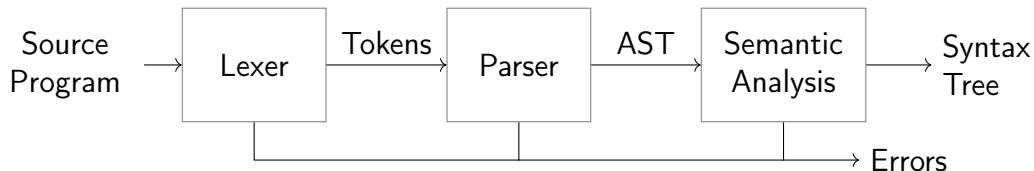
Lecture 2: Compiler Front-end

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2025/26

Compiler Front-end



- ▶ Typical architecture: separate lexer, parser, and context analysis
 - ▶ Allows for more efficient lexical analysis
 - ▶ Smaller components, easier to understand, etc.
- ▶ Some languages: preprocessor and macro expansion

Lexer

- ▶ Convert stream of chars to stream of words (*tokens*)
- ▶ Detect/classify identifiers, numbers, operators, ...
- ▶ Strip whitespace, comments, etc.

$a+b*c \rightarrow \text{ID}(a) \text{ PLUS ID}(b) \text{ TIMES ID}(c)$

- ▶ Typically representable as regular expressions

Typical Token Kinds

- ▶ Punctuators `( ) [ ] { } ; = + += | ||`
- ▶ Identifiers `abc123 main`
- ▶ Keywords `void int __asm__`
- ▶ Numeric constants `123 0xab1 5.7e3 0x1.8p1 09.1f`
- ▶ Char constants `'a' u'œ'`
- ▶ String literals `"abc\x12\n"`
- ▶ Internal `EOF COMMENT UNKNOWN INDENT DEDENT`
 - ▶ Comments might be useful for annotations, e.g. `// fallthrough`

Lexer Implementation

```
struct Token { enum Kind { IDENT, EOF, PLUS, PLUSEQ, /*...*/ };
    std::string_view v; Kind kind; };
Token next(std::string_view v) {
    if (v.empty()) return Token{v, Token::EOF};
    if (v.starts_with("+=")) return Token{"+="sv, Token::PLUSEQ};
    if (v.starts_with("+")) return Token{"+"sv, Token::PLUS};
    switch (v[0]) {
    case ' ', '\n', '\t': return next(v.substr(1)); // skip whitespace
    case 'a' ... 'z', 'A' ... 'Z', '_': {
        Token t = // ... parse identifier, e.g. using regex
        if (auto kind = isKeyword(t.v)) return Token{*kind, t.v};
        return t;
    }
    case '0' ... '9': // ... parse number
    default: return Token{v.substr(0, 1), Token::ERROR};
    }
}
```

Lexing C??=

```
main() <%  
    // yay, this is C99??/  
    puts("hi_world!");  
    puts("what's_up??!");  
%>
```

Output: what's up|

- ▶ Trigraphs for systems with more limited encodings/char sets
- ▶ Digraphs to provide a more readable alternative...

Lexer Implementation

- ▶ Essentially a DFA (for most languages)
 - ▶ Set of regexes $\rightarrow$ NFA $\rightarrow$ DFA
- ▶ Respect whitespace/separators for operators, e.g. + and +=
- ▶ Automatic tools (e.g., flex) exist; most compilers do their own
- ▶ Keywords typically parsed as identifiers first
 - ▶ Check identifier if it is a keyword; can use perfect hashing
- ▶ Other practical problems
 - ▶ UTF-8 homoglyphs; trigraphs; pre-processing directives

Parsing

- ▶ Convert stream of tokens into (abstract) syntax tree
 - ▶ Most programming languages are context-sensitive
 - ▶ Variable declarations, argument count, type match, etc.
 $\rightsquigarrow$ separated into semantic analysis
- Syntactically valid: `void foo = doesntExist / "abc";`
- ▶ Grammar usually specified as CFG

Context-Free Grammar (CFG)

- ▶ Terminals: basic symbols/tokens
- ▶ Non-terminals: syntactic variables
- ▶ Start symbol: non-terminal defining language
- ▶ Productions: non-terminal $\rightarrow$ series of (non-)terminals

$$\begin{aligned} stmt &\rightarrow whileStmt \mid breakStmt \mid exprStmt \\ whileStmt &\rightarrow \mathbf{while} \ (\ expr \) \ stmt \\ breakStmt &\rightarrow \mathbf{break} \ ; \\ exprStmt &\rightarrow expr \ ; \\ expr &\rightarrow expr \ + \ expr \mid expr \ * \ expr \mid expr \ = \ expr \mid (\ expr \) \mid \mathbf{number} \end{aligned}$$

Hand-written Parsing – First Try

- ▶ One function per non-terminal
- ▶ Check expected structure
- ▶ Return AST node
- ▶ Need look-ahead!

```
NodePtr parseBreakStmt() {  
    consume(Token::BREAK);  
    consume(Token::SEMICOLON);  
    return newNode(Node::BreakStmt);  
}  
  
NodePtr parseWhileStmt() {  
    consume(Token::WHILE);  
    consume(Token::LPAREN);  
    NodePtr expr = parseExpr();  
    consume(Token::RPAREN);  
    NodePtr body = parseStmt();  
    return newNode(Node::WhileStmt,  
        {expr, body});  
}  
  
NodePtr parseStmt() {  
    // whoops!  
}
```

Hand-written Parsing – Second Try

- ▶ Need look-ahead to distinguish production rules
 - ▶ Consequences for grammar:
 - ▶ No left-recursion
 - ▶ First n terminals must allow distinguishing rules
 - ▶ $LL(n)$ grammar; n typically 1
- ⇒ Not all CFGs (easily) parseable (but most programming langs. are)
- ▶ Now... expressions

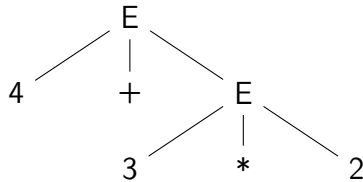
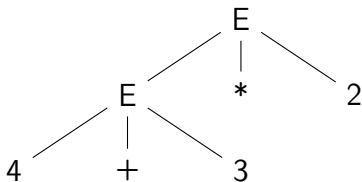
```
NodePtr parseBreakStmt() { /*...*/ }
NodePtr parseWhileStmt() { /*...*/ }

NodePtr parseStmt() {
    Token t = peekToken();
    if (t.kind == Token::BREAK)
        return parseBreakStmt();
    if (t.kind == Token::WHILE)
        return parseWhileStmt();
    // ...
    NodePtr expr = parseExpr();
    consume(Token::SEMICOLON);
    return newNode(Node::ExprStmt,
        {expr});
}
```

Ambiguity

$expr \rightarrow expr + expr \mid expr * expr \mid expr = expr \mid (expr) \mid \text{number}$

Input: $4 + 3 * 2$

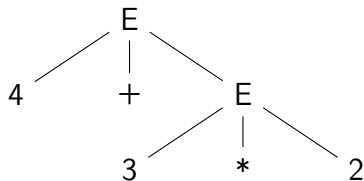


Ambiguity – Rewrite Grammar?

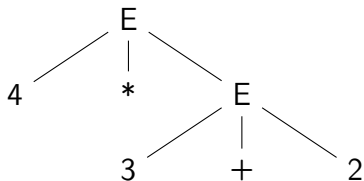
$primary \rightarrow (expr) \mid \mathbf{number}$

$expr \rightarrow primary + expr \mid primary * expr \mid primary = expr \mid primary$

Input: $4 + 3 * 2$

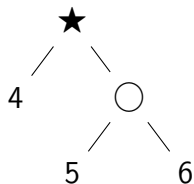


Input: $4 * 3 + 2$

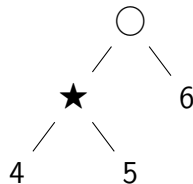


Ambiguity – Precedence

Input: 4 ★ 5 ○ 6



If $\text{prec}(\bigcirc) > \text{prec}(\star)$ or
equal prec. and ★ is right-assoc.



If $\text{prec}(\bigcirc) < \text{prec}(\star)$ or
equal prec. and ★ is left-assoc.

Hand-written Parsing – Expression Parsing

- ▶ Start with basic expr.:
- ▶ Number, variable, etc.
- ▶ Parenthesized expr.
 - ▶ Parse full expression
 - ▶ Next token must be)
- ▶ Unary expr: followed by expr. with higher prec.
 - ▶ - < unary - < [] / ->

```
NodePtr parseExpr(unsigned minPrec=0);
NodePtr parsePrimaryExpr() {
    switch (Token t = next(); t.kind) {
        case Token::IDENT:
            return makeNode(Node::IDENT, t.v);
        case Token::NUMBER: // ...
        case Token::MINUS:
            // Only exprs with high precedence
            return makeNode(Node::UMINUS,
                {parseExpr(UNARY_PREC)});
        case Token::LPAREN: // ...
            // ...
    }
}
```

Hand-written Parsing – Expression Parsing

- ▶ Only allow ops. with higher prec. on the right child
 - ▶ Right-assoc.: allow same
- ▶ Lower prec.: return + insert higher up in the tree

```
OpDesc OPS[] = { // {prec, rassoc}
  [Token::MUL] = {12, false},
  [Token::ADD] = {11, false},
  [Token::EQ]  = {2,  true},
  [Token::QUEST] = {3, true}, // ?:
}
```

```
NodePtr parseExpr(unsigned minPrec=1) {
  auto lhs = parsePrimaryExpr();
  while (auto op = OPS[next().kind];
         op.prec >= minPrec) {
    // ... handle (, [, ?: ...
    auto newPrec = op.rassoc ?
      op.prec : op.prec + 1;
    auto rhs = parseExpr(newPrec);
    lhs = makeNode(op.nodeKind,
      {lhs, rhs});
  }
  return lhs;
}
```

a = 3 * 2 + 1;

a = b + c + d = 1;

a ? 1 : b ? 2 : 3;

Top-down vs. Bottom-up Parsing

Top-down Parsing

- ▶ Start with top rule
- ▶ Every step: choose expansion
- ▶ LL(1) parser
 - ▶ Left-to-right, Leftmost Derivation
- ▶ “Easily” writable by hand
- ▶ Error handling rather simple
- ▶ Covers many prog. languages

Bottom-up Parsing

- ▶ Start with text
- ▶ Reduce to non-terminal
- ▶ LR(1) parser
 - ▶ Left-to-right, Rightmost Derivation
 - ▶ Strict super-set of LL(1)
- ▶ Often: uses parser generator
- ▶ Error handling more complex
- ▶ Covers nearly all prog. languages

Parser Generators

- ▶ Writing parsers by hand can be large effort
- ▶ Parser generators can simplify parser writing a lot
 - ▶ Yacc/Bison, PLY, ANTLR, ...
- ▶ Automatic generation of parser/parsing tables from CFG
 - ▶ Finds ambiguities in the grammar
 - ▶ Lexer often written by hand
- ▶ Used heavily in practice, unless error handling is important

Bison Example – part 1

```
%define api.pure full
%define api.value.type {ASTNode*}
%param { Lexer* lexer }
%code{
static int yylex(ASTNode** lvalp, Lexer* lexer);
}
%token NUMBER
%token WHILE "while"
%token BREAK "break"

// precedence and associativity
%right '='
%left '+'
%left '*'
```

Bison Example – part 2

```
%%
stmt : WHILE '(' expr ')' stmt { $$ = mkNode(WHILE, $1, $2); }
      | BREAK ';'               { $$ = mkNode(BREAK, NULL, NULL); }
      | expr ';'                 { $$ = $1; }
      ;

expr  : expr '+' expr            { $$ = mkNode('+', $1, $2); }
      | expr '*' expr           { $$ = mkNode('*', $1, $2); }
      | expr '=' expr           { $$ = mkNode('=', $1, $2); }
      | '(' expr ')'            { $$ = $1; }
      | NUMBER
      ;

%%

static int yylex(ASTNode** lvalp, Lexer* lexer) {
    /* return next token, or YYEOF/... */ }
```

Parsing in Practice

- ▶ Some use parser generators, e.g. Python
some use hand-written parsers, e.g. GCC, Clang, Swift, Go
- ▶ Optimization of grammar for performance
 - ▶ Rewrite rules to reduce states, etc.
- ▶ Useful error-handling: complex!
 - ▶ Try skipping to next separator, e.g. `;` or `,`
- ▶ Programming languages are not always context-free
 - ▶ C: `foo* bar;`
 - ▶ May need to break separation between lexer and parser

Parsing C++

- ▶ C++ is not context-free (inherited from C): $T * a;$
- ▶ C++ is ambiguous: `Type (a) , b;`
 - ▶ Can be a declaration or a comma expression
- ▶ C++ templates are Turing-complete⁸
- ▶ C++ *parsing* is hence *undecidable*⁹
 - ▶ Template instantiation combined with $C \ T * a$ ambiguity

⁸TL Veldhuizen. *C++ templates are Turing complete*. 2003. .

⁹J Haberman. *Parsing C++ is literally undecidable*. 2013. .

Semantic Analysis

- ▶ Syntactical correctness $\nRightarrow$ correct program
`void foo = doesntExist / ++"abc";`
- ▶ Needs context-sensitive analysis:
 - ▶ Variable existence, storage, accessibility, ...
 - ▶ Function existence, arguments, ...
 - ▶ Operator type compatibility
 - ▶ Attribute allowance
- ▶ Additional type complexity: inference, polymorphism, ...

Semantic Analysis: Scope Checking with AST Walking

- ▶ Idea: walk through AST (in DFS-order) and validate on the way
- ▶ Keep track of scope with declared variables
 - ▶ Might need to keep track of defined types separately

How to implement the scope data structure?

- ▶ For identifiers: check existence and get type
- ▶ For expressions: check types and derive result type
- ▶ For assignment: check lvalue-ness of left side

- ▶ *Might* be possible during AST creation
- ▶ Needs care with built-ins and other special constructs

Semantic Analysis and Post-Parsing Transformations

- ▶ Check for error-prone code patterns
 - ▶ Completeness of `switch`, out-of-range constants, unused variables, ...
- ▶ Check method calls, parameter types
- ▶ Duplicate code for templates
- ▶ Make implicit value conversions explicit
- ▶ Handle attributes: visibility, warnings, etc.
- ▶ Mangle names, split functions (OpenMP), ABI-specific setup, ...
- ▶ Last step: generate IR code

Parsing Performance

Is parsing/front-end performance important?

- ▶ Not necessarily: normal compilers
 - ▶ Some languages (e.g., Rust) need unbounded time *for parsing*
- ▶ Somewhat: JIT compilers
 - ▶ Start-up time is generally noticable
- ▶ Somewhat more: Developer tools
 - ▶ Imagine: waiting for seconds just for updated syntax highlighting
 - ▶ Often uses tricks like incremental updates to parse tree

Data Types

- ▶ Important part of programming languages
- ▶ Might have large variety and compatibility
 - ▶ Numbers, Strings, Arrays, Compound Types (struct/union), Enum, Templates, Functions, Pointers, ...
 - ▶ Class hierarchy, Interfaces, Abstract Classes, ...
 - ▶ Integer/float compatibility, promotion, ...
- ▶ Might have implicit conversions

Data Types: Implementing Classes

- ▶ Simple class/struct: trivial, just bunch of fields
 - ▶ Methods take (pointer to) `this` as implicit parameter
- ▶ Single inheritance: also trivial – extend struct at end
- ▶ Virtual methods: store vtable in object representation
 - ▶ vtable = table of function pointers for virtual methods
 - ▶ Each sub-class has their own vtable
- ▶ Multiple inheritance is much more involved
- ▶ Dynamic casts: needs run-time type information (RTTI)

Recommended Lectures

AD IN2227 “Compiler Constructions” covers parsing/analysis in depth

AD CIT3230000 “Programming Languages” covers dispatching/mixins/...

Compiler Front-end – Summary

- ▶ Lexer splits input into tokens
 - ▶ Essentially Regex-Matching + Keywords; rather simple
- ▶ Parser constructs (abstract) syntax tree from tokens
 - ▶ Top-down vs. bottom-up parsing
 - ▶ Typical: top-down for control flow; bottom-up for expressions
 - ▶ Respect precedence and associativity for operators
- ▶ Semantic analysis ensures meaningful program
- ▶ Some data structures are complex to implement
- ▶ Some programming languages are more difficult to parse

Compiler Front-end – Questions

- ▶ What are typical components of a compiler front-end?
- ▶ What output does the lexer produce?
- ▶ How does a parser disambiguate rules?
- ▶ What is the typical way to handle operator precedence?
- ▶ Why are not all programming languages describable using CFGs?
- ▶ How to implement classes with virtual functions?