

Code Generation for Data Processing

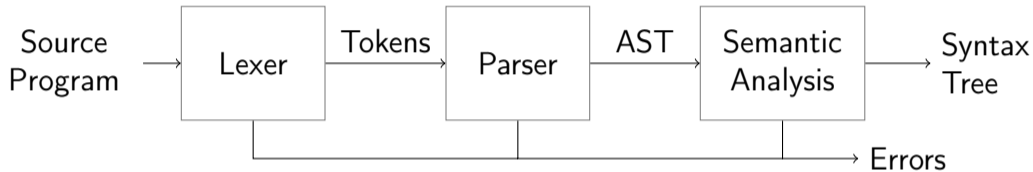
Lecture 2: Compiler Front-end

Alexis Engelke

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

Winter 2022/23

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` → `ID(a) PLUS ID(b) 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`
- ▶ 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

```
def nextToken(inp: str) -> tuple[str, str, str]:
    # Get next token, return (kind, value, remainder)
    inp = inp.lstrip()
    if not inp:
        return "EOF", "", inp
    if inp[0].isdigit():
        m = re.match(r'[1-9][0-9]*|0([0-7]+|x[0-9a-fA-F]+|)', inp)
        return "NUM", m[0], inp[m.end():]
    if inp[0].isalpha():
        m = re.match(r'[a-zA-Z][a-zA-Z0-9_]*', inp)
        if m[0] in KEYWORDS: return m[0], m[0], inp[m.end():]
        return "IDENT", m[0], inp[m.end():]
    if inp[:2] == "+=": return "PLUSEQ", inp[:2], inp[2:]
    if inp[:1] == "+": return "PLUS", inp[:1], inp[1:]
    ...
    raise Exception()
```

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

$stmt \rightarrow whileStmt \mid breakStmt \mid exprStmt$

$whileStmt \rightarrow \mathbf{while} (expr) stmt$

$breakStmt \rightarrow \mathbf{break} ;$

$exprStmt \rightarrow expr ;$

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

Hand-written Parsing – First Try

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

```
def parseBreakStmt(...):  
    matchToken("break")  
    matchToken("SEMICOLON")  
    return ("breakStmt",)
```

```
def parseWhileStmt(...):  
    matchToken("while")  
    matchToken("LPAREN")  
    expr = parseExpr(...)  
    matchToken("RPAREN")  
    stmt = parseStmt(...)  
    return ("whileStmt", expr, stmt)
```

```
def parseStmt(...):  
    # whoops!
```

Hand-written Parsing – Second Try

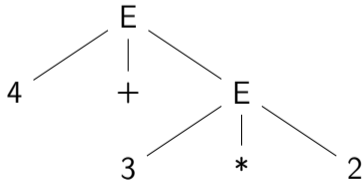
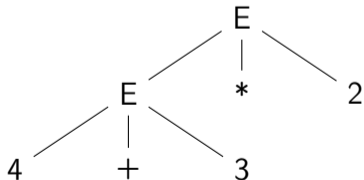
- ▶ Need look-ahead to distinguish production rules
 - ▶ Consequences for grammar:
 - ▶ No left-recursion
 - ▶ First n terminals must allow distinguishing rules
 - ▶ $LL(n)$ grammar; n typically 1
- ⇒ Not all CFGs (easily) parseable (but most programming langs. are)
- ▶ Now... expressions

```
def parseBreakStmt(...):  
    ... # as before  
def parseWhileStmt(...):  
    ... # as before  
  
def parseStmt(...):  
    tok = peekToken()  
    if tok == "break":  
        return parseBreakStmt(...)  
    if tok == "while":  
        return parseWhileStmt(...)  
    expr = parseExpr(...)  
    matchToken("SEMICOLON")  
    return ("exprStmt", expr)
```

Ambiguity

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

Input: $4 + 3 * 2$

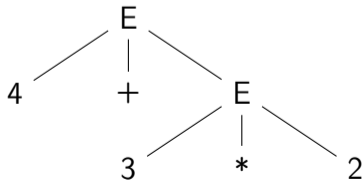


Ambiguity – Rewrite Grammar?

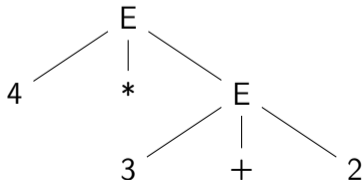
$primary \rightarrow (expr) | \mathbf{number}$

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

Input: $4 + 3 * 2$

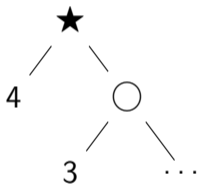


Input: $4 * 3 + 2$

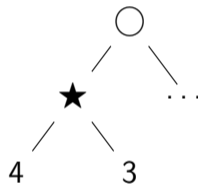


Ambiguity – Precedence

Input: 4 ★ 3 ○ ...



- ▶ $prec(\bigcirc) > prec(\star)$
- ▶ Equal prec. and ★ is right-associative



- ▶ $prec(\bigcirc) < prec(\star)$
- ▶ Equal prec. and ★ is left-associative

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 - < [] /->

```
def parsePrimaryExpr(...):  
    # handle numbers, unary operators,  
    # variables, parenthesized expr.  
    ... # trivial ;)  
def parseExpr(..., minPrec=0):  
    lhs = parsePrimaryExpr(...)  
    ... # (next slide)
```

Hand-written Parsing – Expression Parsing

- ▶ Only allow ops. with higher prec. on the right child
- ▶ Operator precedence
 - ▶ * → (3, left-assoc)
 - ▶ + → (2, left-assoc)
 - ▶ = → (1, right-assoc)
- ▶ Right-assoc.: allow same prec.
 - ▶ Assignment, ternary

```
def parsePrimaryExpr(...):  
    # handle numbers, unary operators,  
    # variables, parenthesized expr.  
    ... # trivial ;)  
def parseExpr(..., minPrec=0):  
    lhs = parsePrimaryExpr(...)  
    while True:  
        tok = nextToken()  
        prec, rassoc = OPERATORS[tok]  
        if prec < minPrec:  
            return lhs  
        # XXX: handling for (, [, ?:  
        newPrec = prec if rassoc else prec+1  
        rhs = parseExpr(..., newPrec)  
        lhs = ("expr", tok, lhs, rhs)
```


Hand-written Parsing – Expression Parsing

```
OPERATORS = {  
    "*": (3, False),  
    "+": (2, False),  
    "=": (1, True),  
}
```

```
def parsePrimaryExpr(...):  
    # handle numbers, unary operators,  
    # variables, parenthesized expr.  
    ... # trivial ;)  
def parseExpr(..., minPrec=0):  
    lhs = parsePrimaryExpr(...)  
    while True:  
        tok = nextToken()  
        prec, rassoc = OPERATORS[tok]  
        if prec < minPrec:  
            return lhs  
        # XXX: handling for: (, [, ?:  
        newPrec = prec if rassoc else prec+1  
        rhs = parseExpr(..., newPrec)  
        lhs = ("expr", tok, lhs, rhs)
```

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
 - ▶ But: lexer often written by hand either way
- ▶ Used heavily in practice

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
- ▶ 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-complete²
- ▶ C++ *parsing* is hence *undecidable*³
 - ▶ Template instantiation combined with C `T * a` ambiguity

²TL Veldhuizen. *C++ templates are Turing complete*. 2003. [🌐](#).

³J Haberman. *Parsing C++ is literally undecidable*. 2013. [🌐](#).

Semantic Analysis

- ▶ Syntactical correctness $\not\Rightarrow$ 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
 - ▶ $Scope = (Map[Name \rightarrow Type] \text{ names}, Scope \text{ parent})$
 - ▶ Might need to keep track of defined types separately
- ▶ 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 noticeable
- ▶ 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: embed all parents, multiple vtables
- ▶ 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/...

Interpretation on the AST



1. Find entry point
2. Walk through AST
 - ▶ Compute values for expressions
 - ▶ Track values of variables in scope
3. Profit!

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?