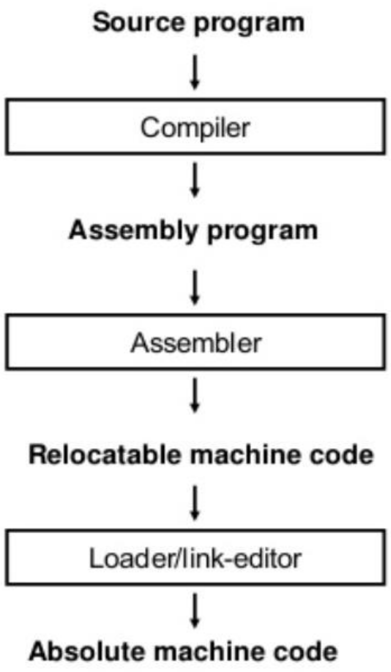
Johns Hopkins Engineering

Module 9: Compilers

EN605.204: Computer Organization

C Function

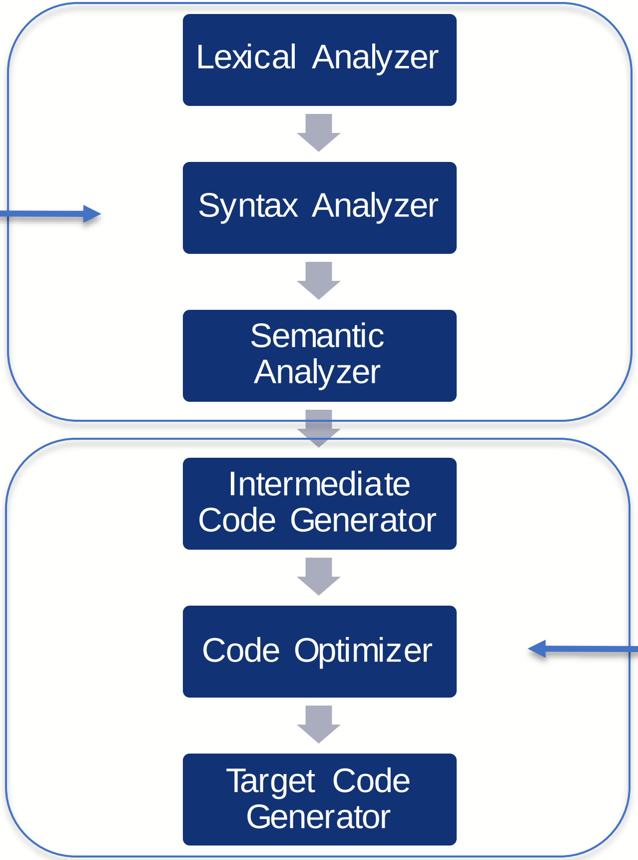
- Computers only speak machine code
- Humans are better at semantic languages
- Compilers translate high-level code from one language (source) to another (target)
  - Ex: C to MIPS
- Decompiling is the process of going from a low-level to high-level code
  - Used to reverse engineer a binary



Compiler Phases

Front-end
(source)

Each step provides input
to each subsequent step



Back-end
(target)

Lexical Analysis

- "lexicon": the set of units in a language
  - Integers: 0-9
  - English: a-z, A-Z, punctuation
- Scanner (lexer) tokenizes (groups) a language's characters
  - Token: collection of units that has meaning in the language
    - Ex: *`int i = 0;`* = 'i', 'n', 't', 'i', '=', '0', ';' = [int, i, =, 0, ;]
- Lexical errors: result in tokens not supported by the language:
  - *`double d }= 3.14159;`* - *= is an invalid token in C

Lexical Analysis

- "lexicon": the set of units in a language
  - Integers: 0-9
  - English: a-z, A-Z, punctuation
- Scanner (lexer) tokenizes (groups) a language's characters
  - Token: collection of units that has meaning in the language
    - Ex: *`int i = 0;`* = 'i', 'n', 't', 'i', '=', '0', ';' = [int, i, =, 0, ;]
- Lexical errors: result in tokens not supported by the language:
  - *`double d }= 3.14159;`* - *= is an invalid token in C

Tokenizing with Regular Expressions

- Most lexical analyzers verify tokens through the use of "regular expressions" (RE's)
- RegEx's are a well-defined syntax for performing string matching
  - Ex: determine whether a password contains 1 lower, 1 upper, and 8+ chars.
    - PASSWORDd_1234
    - Pass_worD!!
- Support for RegEx's in all major languages

RegEx Examples

RegEx	Match	Non-match
^A	Alan	aroma
ab{2}	babble	dribble
fo{2}[dt]	foothill, foodie	fools
be*ar	boar, beard	beat
y(ea)+	yearly	yes
t..th	truth, teeth, tth	teeeth
be{1,2}[15]+$	Ggbe1, bebe5	Bebe5s, 1be5!

^	Start of a string.
$	End of a string.
.	Any character (except \n newline)
|	Alternation.
{...}	Explicit quantifier notation.
[...]	Explicit set of characters to match.
(...)	Logical grouping of part of an expression.
*	0 or more of previous expression.
+	1 or more of previous expression.

RegEx Hint

- Best way to build a RegEx is to say it in plain English:
- Ex: MIPS comment: "begins with a single # and end with 0 to 79 characters that are either lowercase letters, uppercase letters, numbers, or whitespace"
- $#[a-zA-Z0-9\s]{0,79}^

ANSI C Grammar (well, ~10% of it!)

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <compound-statement>

<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef

<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union
```

Syntax Analysis

- Parser builds a tree from the tokens provided by the scanner
- Ensures tokens make sense in context with one another
  - `int x = 0;`
    - Valid lexically, syntactically
  - `int = x 0;`
    - Valid lexically, not syntactically
- How to determine valid syntax between tokens: grammar

Grammar

- Set of rules that define legal syntax of a language
- We'll look at Backus-Naur Form
  - Terminals: literal symbols that are permitted in code
  - Non-terminals: are substituted with other elements
  - Production Rules: how (non) terminals are used together
- Sample grammar:
  - `<digit> ::= '0' | '1' | '2' | '3'`
  - `<integer> ::= ['-'] <digit> {<digit>}`
- This grammar specifies how we can represent integers
  - Valid: 1, -10, 0023, 1230321
  - Invalid: +1, 4, -90, 1234321

Advanced Grammar

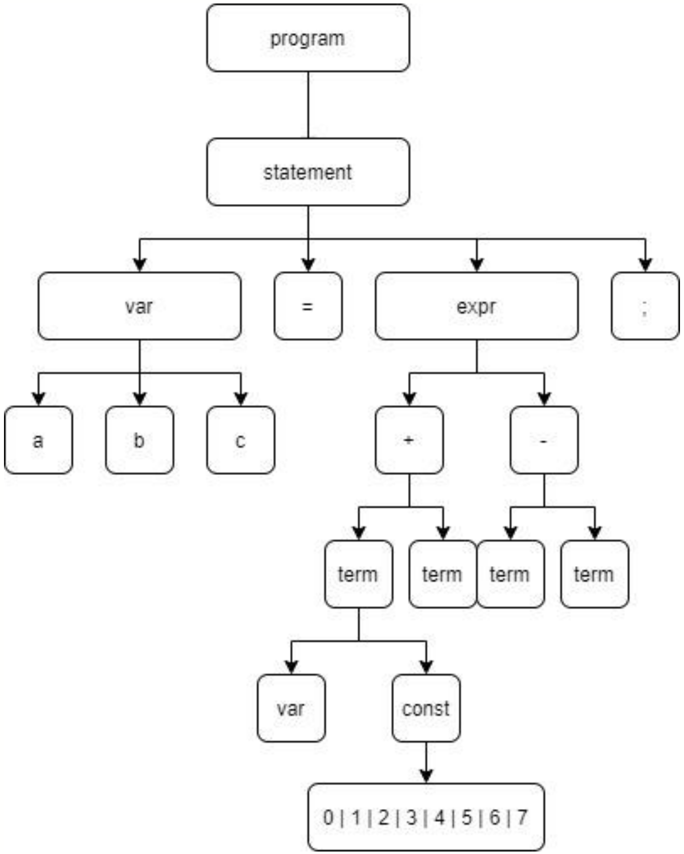
- <program> ::= {<stmt>*}
- <stmt> ::= <var> **=** <expr> ;
- <var> ::= a | b | c
- <expr> ::= <term> + <term> | <term> **-** <term>
- <term> = <var> | <const>
- <const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

- Valid: `a = b + c; c = a + 2;`
- Invalid: `a = b + 8; a = 2 * c;`

terminals

Parse Tree

- Recursively tracing through the grammar results in our "parse tree"
- 'program' contains one 'statement'
- 'stmt' contains 'var', 'expr', '=', and ';'
- 'var' contains a literal 'a', 'b', or 'c'
- 'expr' contains 2 <term>'s separated by a literal '+' or '-'
- 'const' contains a literal '1', '2', '3', '4', '5', '6', or '7'



Semantic Analysis

- Given the parse tree from the parser, makes sure the "meaning" of the code makes sense
- Static checks
  - Type checking: int x should not = double d
    - Maintains symbol table mapping variables to types
  - Initialization: C requires variables to be set to 0 before use
- Dynamic checks (checked at runtime, code provided by compiler)
  - Cannot use a variable before declaring it
  - Java checks array bounds