Introduction to Compilers

Introduction to Course

1. Study compilers which translate a high-level source language to a lower-level assembly code
    - Focus on theories and algorithms that can be applied to any language or machine
2. Course is divided into three sections
    - Front end: Deals with syntactic and semantic analysis of the source language
    - Middle end: Deals with the intermediate representation, its analysis, and optimizations into which the source code is translated
    - Back end: Code generator that generates machien code

Introduction to Compilers

1. Study the following topics:
    - Tokenization
    - Parsing
    - Symbol tables
    - Semantic analysis

What is a Compiler?

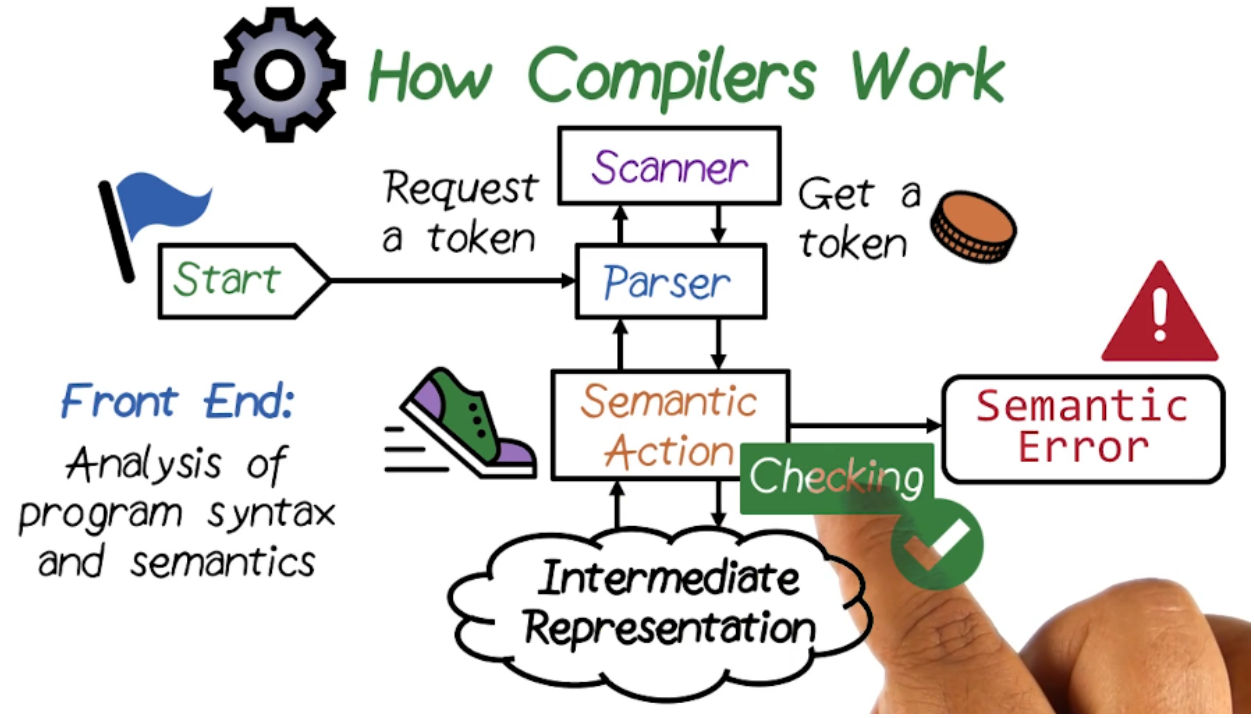
1. Compiler: A program that translates a program from a source language into a target language
    - High-level: C, C++, Java, Fortran
    - Low-level: Machine code executed in binary form on the processor
2. Interpreter vs compiler
    - Interpreter works off same source files
    - Instead of translating it into an executable, it interprets the program line by line and displays the results on the screen
    - Compiled code is much faster than interpretation

Why Compilers?

1. IBM created the first compiler in 1954
    - Ratfor, used for compiling Fortran
2. Compilers allow us to use higher level abstractions, like arrays, to program software
    - This allows us to iterate very quickly
        - Programming in assembly would be much slower and more error prone
    - Assembler converts assembly to bits executed directly on processor
        - Job of an assembler is much easier than a compiler

How Compilers Work: Overview

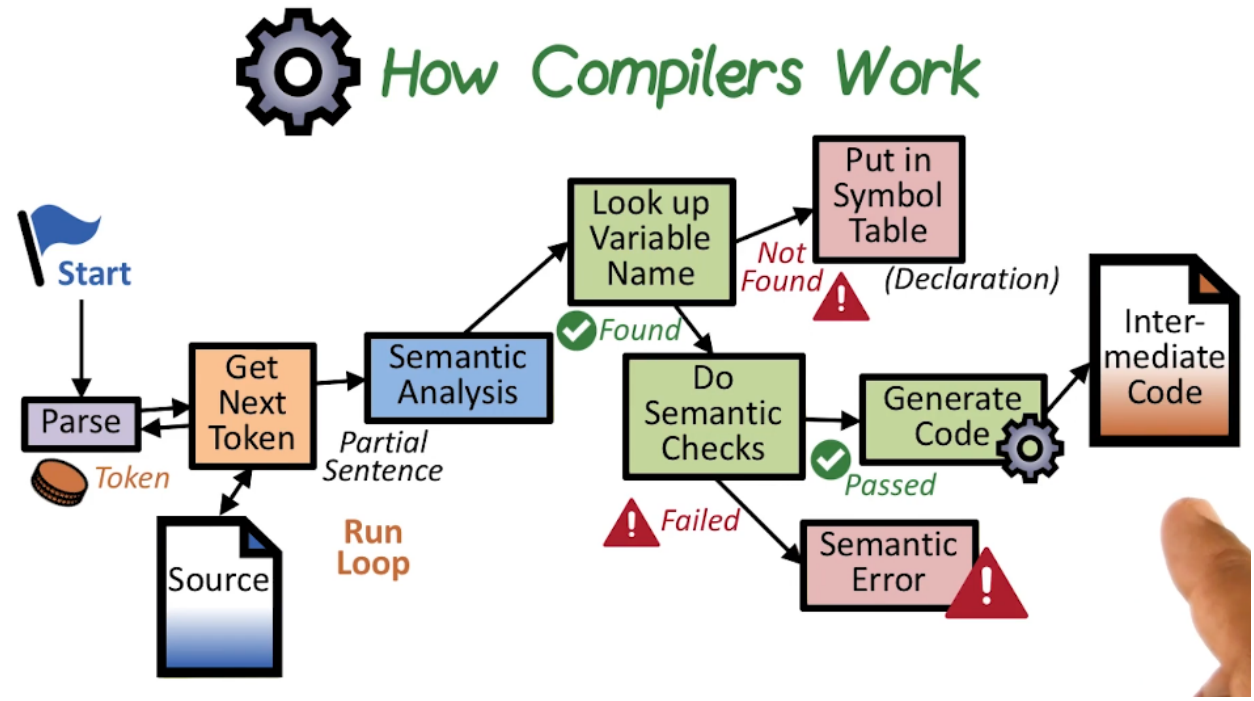
1. Parser: Requests a token from the scanner
    - Checks if the token is syntactically correct
2. Scanner: Gives a token to the parser
    - Splits program along demarcation
    - 157.699 would be parsed into a FLOATCONST with value 157.699
3. Semantic action: Parser ensures words returned by scanner are semantically correct
4. Intermediate representation: Take this high level language construct and convert it into a form which is closer to the assembly
5. Parser, scanner, and semantic analysis are the front-end of the compiler
    - Front end is highly customized to the language syntax and semantics



Compiler Overview

How Compilers Work: Details

1. Parser tracks where we are in the compilation
   - How much of the program are we seeing?
   - What do we do with the program?
   - Can we do some checks on the program?
   - Can we undertake some translation on the program?
   - Parser is one of the key phases of the compiler
2. Parser invokes scanner and asks for next token
   - Parser and scanner continue for many iterations
   - Get next token, check for syntactic correctness, get next token
3. Semantic analysis
   - Look up variable name:
     - Have you declared a variable before using it?
     - Is it of a particular type?
     - Is it within the current scope?
   - Do semantic checks
     - If you are processing an addition sentence and one operand is a character, this could be a semantic error
   - If a variable isn't found, it will place it in the symbol table
     - Will use symbol table for future references
   - If semantic checks pass, we will generate code
     - Not assembly, intermediate representation
     - Next pass will generate the assembly
   - Semantic errors:
     - Undeclared variable
     - Declared variable, but not available in current scope
     - Incompatible types



Compiler Details

Compiler Parts

1. Front End: Analysis
   - Lexical analysis: Scan and group characters into words
   - Syntax analysis: Parsing to find out if syntax of statement is correct
   - Semantic analysis: Find out if the statement is semantically meaningful
2. Back End: Synthesis
   - Code generation: Convert intermediate representation into assembly
   - Optimization: Make the code efficient to improve the speed of the code

The Big Picture

1. Scanning: Converting input text into stream of known objects called tokens
   - Simplifies parsing process
2. Parsing: Translating code to rules of grammar
   - Building representation of code
3. Lexical rules of language dictate how legal word is formed by concatenating alphabet
4. Grammar dictates syntactic rules of language
   - i.e., how legal sentence could be formed
   - Assignment should have left hand side, equals sign, expression, semicolon

Tokenization Quiz

1. If we use a space to denote the end of a word, how many tokens are in the following sentence in the quotes? (quotes and period to be excluded)
   - "The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."
   - 21 words

Scanning and Tokenization

1. Scanner reads a character at a time from the file
   - Puts these tokens one by one into the token buffer
   - Token buffer contains: token being identified

```
#include <stdio.h>
int main()
{
    printf("Hello world");
}
```

2. Scanner will read "main" then "("
   - At this point, the scanner knows main(is not part of C's language lexicon
   - It will send main to the parser and start parsing the next token
   - Go until the token is no longer valid and assume the next character is part of the next token
   - Longest match algorithm is performed by all scanners

Parser Quiz

1. Grammar rules:
   - E -> E + E
   - E -> E * E
   - E -> -E
   - E -> (E)
   - E -> id (variable name)
   - Parsing: Applying the rules to check if the expression is valid or not
     - Example: (E)) is an invalid expression
     - Example: E ** E is an invalid expression
2. Check all the statements that are grammatically correct give the grammar rules.
   - a + b (Correct)
   - a + b * c (Correct)
   - a b + c (Incorrect)
   - a + b + (a) c (Incorrect)

Parser

1. Parser: Check the syntax using the grammatical rules of the language
2. Control the overall operation
3. Demands scanner to produce a token
4. Outcomes
   - Failure: Syntax error
   - Success: Does nothing and returns to get next token, or takes semantic action

Parser: Grammar Rules

1. Grammar for a micro C language
   - <C-PROG> -> MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>
   - <PARAMS> -> NULL
   - <PARAMS> -> VAR <VARLIST>
   - <VARLIST> -> , VAR <VARLIST>
   - <VARLIST> -> NULL
   - <MAIN-BODY> -> CURLYOPEN <DECL-STMT> <ASSIGN-STMT> CURLYCLOSE
   - <DECL-STMT> -> <TYPE> VAR <VARLIST>
   - <ASSIGN-STMT> -> VAR = <EXPR>;
   - <EXPR> -> VAR

- <EXPR> -> VAR <OP> <EXPR>
- <OP> -> +
- <OP> -> *
- <TYPE> -> INT
- <TYPE> -> FLOAT

2. This grammar does not specify mixing ints/floats
   - Part of the semantic specification
     - Semantic check, not syntactic check
   - Checking syntax is much cheaper than checking semantics
     - Do the cheap things first and the complex things later
     - Don't check semantics if syntax is wrong

Parser: Example

```
main()
{
    int a, b;
    a = b;
}
```

Parser: Overview

1. How does the parser work?
   - Start matching using a rule
   - When match takes place at certain position, move further (get next token and repeat)
   - If expansion needs to be done, choose appropriate rule (How to decide which rule to choose?)
   - If no rule found, declare error
   - If several rules are found, the grammar (set of rules) is ambiguous - something wrong with it!

Parser: Ambiguity

1. Ambiguity can lead to understanding a given sentence in two different ways which allows assigning two meanings to the same sentence, some thing highly undesirable, e.g. 2 * 2 + 3 can be interpreted as understanding 2 to be multiplied by 2 and the result to be added to 3 yielding 7...
2. Or 2 and 3 to be added together first and then multiply 2 to the result yielding 10
   - Answer changes depending on the order that the rules are applied

Ambiguous Parsing Quiz

1. Given:
   - if x == 1 then if y == 2 print 1 else print 2
2. Which grammar rule should be applied?
   - stmt -> if expr then stmt (Correct)
   - stmt -> if expr then stmt else stmt (Correct)
3. Either statement can apply and the output will change depending on how it is applied
   - To resolve ambiguity, always associate else with innermost if

```
// program 1
if (x == 1) {
    if (y == 2) {
        printf("1");
    } else {
        printf("2");
    }
}
```

```c
// program 2
if (x == 1) {
    if (y == 2) {
        printf("1");
    }
} else {
    printf("2");
}
```

```
stmt           → matched_stmt | unmatched_stmt / other_stmt
matched_stmt   → if expr then matched_stmt else matched_stmt
                 | other_stmt
unmatched_stmt → if expr then stmt
                 | if expr then matched_stmt else unmatched_stmt
```

Unambiguous Parsing Grammar

Scanning and Parsing

1. Tokens for example program
   - MAIN
   - OPENPAR
   - NULL (implicit; could be VAR, but isn't in this case)
   - CLOSEPAR
   - CURLYOPEN
   - INT
   - VAR
   - COMMA ","
   - VAR
   - SEMICOLON ";"
   - VAR
   - ASSIGN "="
   - VAR
   - SEMICOLON ";"
   - CURLYCLOSE

Syntax vs Semantics Quiz

1. Assuming we are using 'C' programming language, determine which statements are syntactically incorrect (A), which are semantically incorrect (B), and which are both syntactically and semantically incorrect (C), and which are correct (D).

```c
int main()
{
    int a, b, c;
    a +- = b; // A
    a = b + c; // D
    d = a + b + ; // C
```

```
    d = a + b; // B
}
```

Syntax vs Semantics

1. During Parsing? (syntax checking)
   - Tokens are matched and consumed by parser
   - Symbol tables
   - Variables have attributes
   - Declaration attached attribtues to variables
2. Semantic checks
   - What are semantic actions?

Symbol Table

1. int a,b; -> Declares a and b, within current scope, type integer
   - Use of a and b now legal
2. Basic Symbol Table

Name	Type	Scope
a	int	main
b	int	main

Semantic Actions: Overview

1. Enter variable declaration into symbol table
2. Look up variables in symbol table
3. Do binding of looked-up variables (scoping rules, etc.)
4. Do type checking for compatibility
5. Keep the semantic context of processing
   - a + b + c
   - t1 = a + b
   - t2 = t1 + c
   - We track the types of the temporaries to ensure compatibility

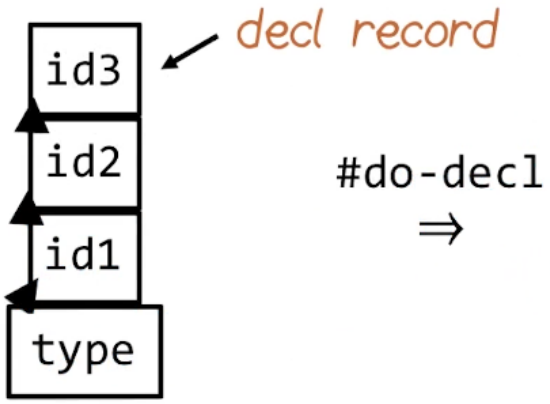
Semantic Actions: Implementation

1. Action symbols embedded in the grammar.
   - Each action symbol represents a semantic procedure
   - These procedures do things and/or return values
2. Semantic procedures are called by parser at appropriate places during parsing
   - Once two operands and an operator are parsed, we can call a semantic procedure on these to check for scope, type compatibility, etc.
3. Semantic stack implements and stores semantic records

Semantic Actions: Symbols

1. put-type: puts given type on semantic stack
2. proc-decl: builds decl record for var on stack
3. add-decl: builds decl chain
4. do-decl: traverses chain on semantic stack using backwards pointers entering each var into symbol table

Semantic Actions: Example



Name	Type	Scope
id1	1	3
id2	1	3
id3	1	3

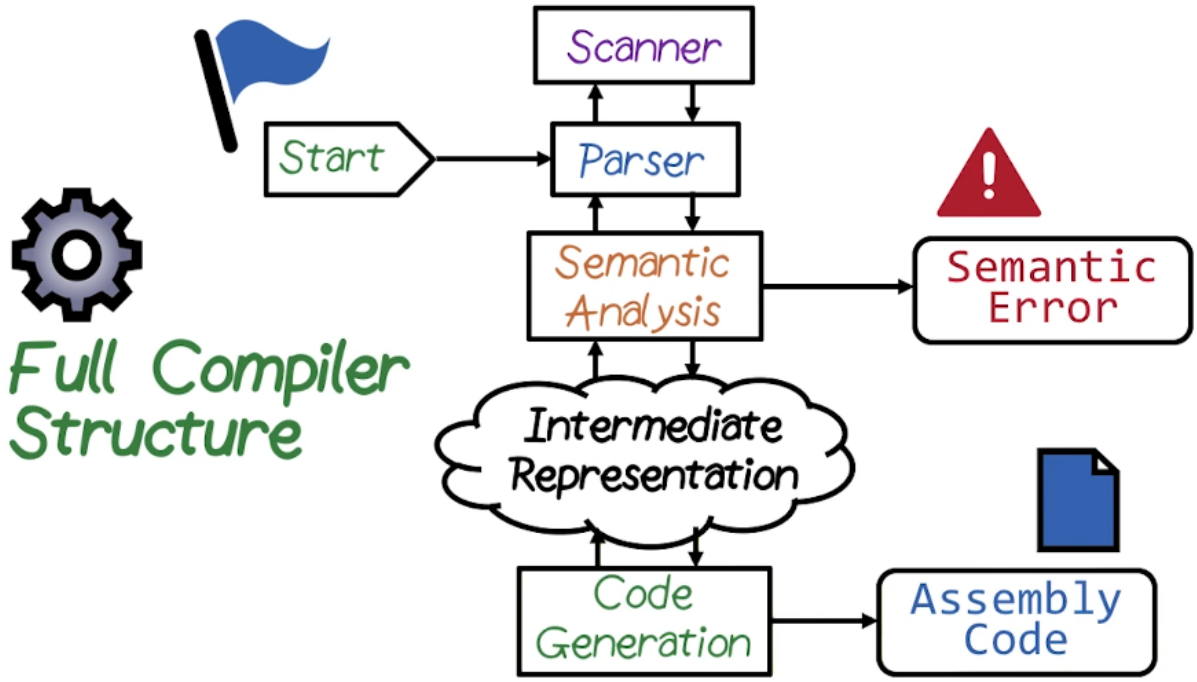
Semantic Actions

Semantic Actions: Two Types

1. Two types of semantic actions:
   - Checking: binding, type compatibility, scoping, etc.
   - Translation: generate temporary values, propagate them to keep semantic context

Full Compiler Structure

1. Front end handles all the parsing and semantics
   - Output is intermediate representation
2. Back end handles code generation
   - Input is intermediate representation
   - Steps:
     - Optimization
     - Register allocation
     - Instruction selection
     - Instruction scheduling



Compiler Structure