The Structure of a Compiler

A compiler performs two major tasks:

- Analysis of the source program being compiled
- Synthesis of a target program

Almost all modern compilers are *syntax-directed:* The compilation process is driven by the syntactic structure of the source program.

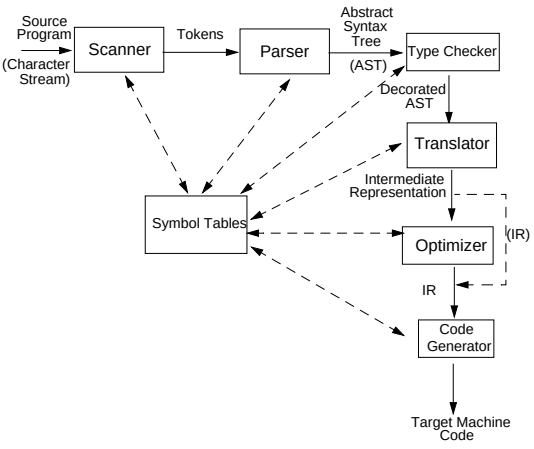
A parser builds semantic structure out of tokens, the elementary symbols of programming language syntax. Recognition of syntactic structure is a major part of the analysis task.

Semantic analysis examines the meaning (semantics) of the program. Semantic analysis plays a dual role.

It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). Semantic analysis also begins the synthesis phase.

The synthesis phase may translate source programs into some intermediate representation (IR) or it may directly generate target code.

If an IR is generated, it then serves as input to a *code generator* component that produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated.



The Structure of a Syntax-Directed Compiler

Scanner

The scanner reads the source program, character by character. It groups individual characters into tokens (identifiers, integers, reserved words, delimiters, and so on). When necessary, the actual character string comprising the token is also passed along for use by the semantic phases.

The scanner:

- Puts the program into a compact and uniform format (a stream of tokens).
- Eliminates unneeded information (such as comments).
- Sometimes enters preliminary information into symbol tables (for example, to register the presence of a particular label or identifier).
- Optionally formats and lists the source program

Building tokens is driven by token descriptions defined using *regular expression* notation.

Regular expressions are a formal notation able to describe the tokens used in modern programming languages. Moreover, they can drive the *automatic generation* of working scanners given only a specification of the tokens. Scanner generators (like Lex, Flex and Jlex) are valuable compiler-building tools.

Parser

Given a syntax specification (as a context-free grammar, CFG), the parser reads tokens and groups them into language structures.

Parsers are typically created from a CFG using a parser generator (like Yacc, Bison or Java CUP).

The parser verifies correct syntax and may issue a syntax error message.

As syntactic structure is recognized, the parser usually builds an abstract syntax tree (AST), a concise representation of program structure, which guides semantic processing.

Type Checker (Semantic Analysis)

The type checker checks the *static semantics* of each AST node. It verifies that the construct is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on).

If the construct is semantically correct, the type checker "decorates" the AST node, adding type or symbol table information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler's target machine.

Translator (Program Synthesis)

If an AST node is semantically correct, it can be translated. Translation involves capturing the run-time "meaning" of a construct.

For example, an AST for a while loop contains two subtrees, one for the loop's control expression, and the other for the loop's body. *Nothing* in the AST shows that a while loop loops! This "meaning" is captured when a while loop's AST is translated. In the IR, the notion of testing the value of the loop control expression,

and conditionally executing the loop body becomes explicit.

The translator is dictated by the semantics of the source language. Little of the nature of the target machine need be made evident. Detailed information on the nature of the target machine (operations available, addressing, register characteristics, etc.) is reserved for the code generation phase.

In simple non-optimizing compilers (like our class project), the translator generates target code directly, without using an IR.

More elaborate compilers may first generate a high-level IR

(that is source language oriented) and then subsequently translate it into a low-level IR (that is target machine oriented). This approach allows a cleaner separation of source and target dependencies.

Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the optimizer.

The term optimization is misleading: we don't always produce the best possible translation of a program, even after optimization by the best of compilers.

Why?

Some optimizations are *impossible* to do in all circumstances because they involve an undecidable problem. Eliminating unreachable ("dead") code is, in general, impossible.

Other optimizations are too expensive to do in all cases. These involve NP-complete problems, believed to be inherently exponential. Assigning registers to variables is an example of an NP-complete problem.

Optimization can be complex; it may involve numerous subphases, which may need to be applied more than once.

Optimizations may be turned off to speed translation. Nonetheless, a well designed optimizer can significantly speed program execution by simplifying, moving or eliminating unneeded computations.

Code Generator

IR code produced by the translator is mapped into target machine code by the code generator. This phase uses detailed information about the target machine and includes machine-specific optimizations like *register allocation* and *code scheduling*.

Code generators can be quite complex since good target code requires consideration of many special cases.

Automatic generation of code generators is possible. The basic approach is to match a low-level IR to target instruction templates, choosing

instructions which best match each IR instruction.

A well-known compiler using automatic code generation techniques is the GNU C compiler. GCC is a heavily optimizing compiler with machine description files for over ten popular computer architectures, and at least two language front ends (C and C++).

Symbol Tables

A symbol table allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is used, a symbol table provides access to the information collected about the identifier when its declaration was processed.

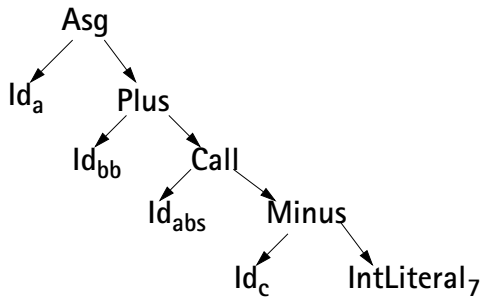
Example

Our source language will be *CSX*, a blend of C, C++ and Java.

Our target language will be the Java JVM, using the Jasmin assembler.

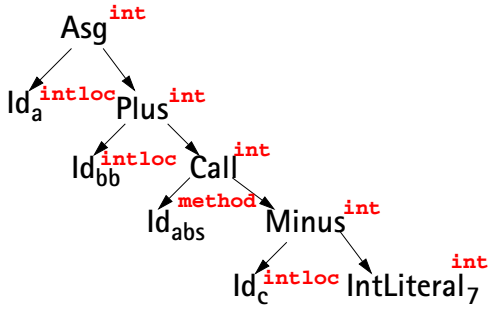
- Our source line is
  `a = bb+abs(c-7);`
  this is a sequence of ASCII characters in a text file.
- The scanner groups characters into tokens, the basic units of a program.
  `a = bb+abs(c-7);`
  After scanning, we have the following token sequence:
  Id_a Asg Id_{bb} Plus Id_{abs} Lparen Id_c Minus
  $IntLiteral_7$ Rparen Semi

- The parser groups these tokens into language constructs (expressions, statements, declarations, etc.) represented in tree form:



(What happened to the parentheses and the semicolon?)

- The type checker resolves types and binds declarations within scopes:



- Finally, JVM code is generated for each node in the tree (leaves first, then roots):

```
iload  3   ; push local 3 (bb)
iload  2   ; push local 2 (c)
ldc    7   ; Push literal 7
isub       ; compute c-7
invokestatic  java/lang/Math/
abs(I)I
iadd       ; compute bb+abs(c-7)
istore  1  ; store result into
             local 1(a)
```

Interpreters

There are two different kinds of interpreters that support execution of programs, *machine interpreters* and *language interpreters*.

Machine Interpreters

Machine interpreters simulate the execution of a program compiled for a particular machine architecture. Java uses a *bytecode interpreter* to simulate the effects of programs compiled for the JVM. Programs like SPIM simulate the execution of a MIPS program on a non-MIPS computer.

Language Interpreters

Language interpreters simulate the effect of executing a program without compiling it to any particular instruction set (real or virtual). Instead some IR form (perhaps an AST) is used to drive execution.

Interpreters provide a number of capabilities not found in compilers:

- Programs may be modified as execution proceeds. This provides a straightforward interactive debugging capability. Depending on program structure, program modifications may require reparsing or repeated semantic analysis. In Python, for example, any string variable may be interpreted as a Python expression or statement and executed.

- Interpreters readily support languages in which the type of a variable denotes may change dynamically (e.g., Python or Scheme). The user program is continuously reexamined as execution proceeds, so symbols need not have a fixed type. Fluid bindings are much more troublesome for compilers, since dynamic changes in the type of a symbol make direct translation into machine code difficult or impossible.
- Interpreters provide better diagnostics. Source text analysis is intermixed with program execution, so especially good diagnostics are available, along with interactive debugging.
- Interpreters support machine independence. All operations are performed within the interpreter. To move to a new machine, we just recompile the interpreter.

However, interpretation can involve large overheads:

- As execution proceeds, program text is continuously reexamined, with bindings, types, and operations sometimes recomputed at each use. For very dynamic languages this can represent a 100:1 (or worse) factor in execution speed over compiled code. For more static languages (such as C or Java), the speed degradation is closer to 10:1.
- Startup time for small programs is slowed, since the interpreter must be load and the program partially recompiled before execution begins.

- Substantial space overhead may be involved. The interpreter and all support routines must usually be kept available. Source text is often not as compact as if it were compiled. This size penalty may lead to restrictions in the size of programs. Programs beyond these built-in limits cannot be handled by the interpreter.

Of course, many languages (including, C, C++ and Java) have both interpreters (for debugging and program development) and compilers (for production work).