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.

Reading Assignment

Read Chapter 3 of Crafting a Compiler.
**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.

- A simple source line is
  \[ a = bb + \text{abs}(c - 7); \]
  this is a sequence of ASCII characters in a text file.
- The scanner groups characters into tokens, the basic units of a program.
  \[ a = bb + \text{abs}(c - 7); \]
  After scanning, we have the following token sequence:
  \[ \text{Id}_a \text{ Asg Id}_{bb} \text{ Plus Id}_{\text{abs}} \text{ Lparen Id}_c \text{ Minus Id}_{\text{abs}} \text{ Lparen Id}_c \text{ IntLiteral} \text{ Rparen Semi} \]

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

  ![Tree Diagram]

  (What happened to the parentheses and the semicolon?)
• The type checker resolves types and binds declarations within scopes:

```
Asg
  \int
Id_a
  Plus
  \int
  \intloc
Id_bb
Call
  \int
  \method
Id_abs
Minus
  \int
  \intloc
Id_c
  \int
IntLiteral
```

• 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)
```

---

Symbol Tables & Scoping

Programming languages use scopes to limit the range in which an identifier is active (and visible).

Within a scope a name may be defined only once (though overloading may be allowed).

A symbol table (or dictionary) is commonly used to collect all the definitions that appear within a scope.

At the start of a scope, the symbol table is empty. At the end of a scope, all declarations within that scope are available within the symbol table.

A language definition may or may not allow forward references to an identifier.

If forward references are allowed, you may use a name that is defined later in the scope (Java does this for field and method declarations within a class).

If forward references are not allowed, an identifier is visible only after its declaration. C, C++, and Java do this for variable declarations.

In CSX, no forward references are allowed.

In terms of symbol tables, forward references require two passes over a scope. First all
declarations are gathered. Next, all references are resolved using the complete set of declarations stored in the symbol table. If forward references are disallowed, one pass through a scope suffices, processing declarations and uses of identifiers together.

Block Structured Languages
- Introduced by Algol 60, includes C, C++, CSX and Java.
- Identifiers may have a non-global scope. Declarations may be local to a class, subprogram or block.
- Scopes may nest, with declarations propagating to inner (contained) scopes.
- The lexically nearest declaration of an identifier is bound to uses of that identifier.

Example (drawn from C):

```c
int x,z;
void A() {
    float x,y;
    print(x,y,z);
}
void B() {
    print(x,y,z)
}
```

Block Structure Concepts
- Nested Visibility
  No access to identifiers outside their scope.
- Nearest Declaration Applies
  Using static nesting of scopes.
- Automatic Allocation and Deallocation of Locals
  Lifetime of data objects is bound to the scope of the identifiers that denote them.
Is Case Significant?

In some languages (C, C++, Java and many others) case is significant in identifiers. This means `aa` and `AA` are different symbols that may have entirely different definitions.

In other languages (Pascal, Ada, Scheme, CSX) case is not significant. In such languages `aa` and `AA` are two alternative spellings of the same identifier.

Data structures commonly used to implement symbol tables usually treat different cases as different symbols. This is fine when case is significant in a language. When case is insignificant, you probably will need to *strip case* before entering or looking up identifiers.

This just means that identifiers are converted to a uniform case before they are entered or looked up. Thus if we choose to use lower case uniformly, the identifiers `aaa`, `AAA`, and `AaA` are all converted to `aaa` for purposes of insertion or lookup.

BUT, inside the symbol table the identifier is stored in the form it was declared so that programmers see the form of identifier they expect in listings, error messages, etc.

How are Symbol Tables Implemented?

There are a number of data structures that can reasonably be used to implement a symbol table:

- **An Ordered List**
  Symbols are stored in a linked list, sorted by the symbol's name. This is simple, but may be a bit too slow if many identifiers appear in a scope.

- **A Binary Search Tree**
  Lookup is much faster than in linked lists, but rebalancing may be needed. (Entering identifiers in sorted order turns a search tree into a linked list.)

- **Hash Tables**
  The most popular choice.

Implementing Block-Structured Symbol Tables

To implement a block structured symbol table we need to be able to efficiently open and close individual scopes, and limit insertion to the innermost current scope.

This can be done using one symbol table structure if we tag individual entries with a “scope number.”

It is far easier (but more wasteful of space) to allocate one symbol table for each scope. Open scopes are stacked, pushing and popping tables as scopes are opened and closed.
Be careful though—many preprogrammed stack implementations don’t allow you to "peek" at entries below the stack top. This is necessary to lookup an identifier in all open scopes.

If a suitable stack implementation (with a peek operation) isn’t available, a linked list of symbol tables will suffice.

**Scanning**

A scanner transforms a character stream into a token stream.

A scanner is sometimes called a lexical analyzer or lexer.

Scanners use a formal notation (regular expressions) to specify the precise structure of tokens. But why bother? Aren’t tokens very simple in structure?

Token structure can be more detailed and subtle than one might expect. Consider simple quoted strings in C, C++ or Java. The body of a string can be any sequence of characters except a quote character (which must be escaped). But is this simple definition really correct?

Can a newline character appear in a string? In C it cannot, unless it is escaped with a backslash.

C, C++ and Java allow escaped newlines in strings, Pascal forbids them entirely. Ada forbids all unprintable characters.

Are null strings (zero-length) allowed? In C, C++, Java and Ada they are, but Pascal forbids them. (In Pascal a string is a packed array of characters, and zero length arrays are disallowed.)

A precise definition of tokens can ensure that lexical rules are clearly stated and properly enforced.

**Regular Expressions**

Regular expressions specify simple (possibly infinite) sets of strings. Regular expressions routinely specify the tokens used in programming languages.

Regular expressions can drive a scanner generator.

Regular expressions are widely used in computer utilities:

- The Unix utility `grep` uses regular expressions to define search patterns in files.
- Unix shells allow regular expressions in file lists for a command.
• Most editors provide a “context search” command that specifies desired matches using regular expressions.
• The Windows Find utility allows some regular expressions.

Regular Sets
The sets of strings defined by regular expressions are called regular sets. When scanning, a token class will be a regular set, whose structure is defined by a regular expression. Particular instances of a token class are sometimes called lexemes, though we will simply call a string in a token class an instance of that token. Thus we call the string abc an identifier if it matches the regular expression that defines valid identifier tokens.

Regular expressions use a finite character set, or vocabulary (denoted $\Sigma$).

This vocabulary is normally the character set used by a computer. Today, the ASCII character set, which contains a total of 128 characters, is very widely used. Java uses the Unicode character set which includes all the ASCII characters as well as a wide variety of other characters.

An empty or null string is allowed (denoted $\lambda$, “lambda”). Lambda represents an empty buffer in which no characters have yet been matched. It also represents optional parts of tokens. An integer literal may begin with a plus or minus, or it may begin with $\lambda$ if it is unsigned.

Catenation
Strings are built from characters in the character set $\Sigma$ via catenation.

As characters are catenated to a string, it grows in length. The string do is built by first catenating d to $\lambda$, and then catenating o to the string d. The null string, when catenated with any string s, yields s. That is, $s \lambda \equiv \lambda \cdot s \equiv s$. Catenating $\lambda$ to a string is like adding 0 to an integer—nothing changes.

Catenation is extended to sets of strings:
Let P and Q be sets of strings. (The symbol $\in$ represents set membership.) If $s_1 \in P$ and $s_2 \in Q$ then string $s_1s_2 \in (P \cup Q)$.
Alternation

Small finite sets are conveniently represented by listing their elements. Parentheses delimit expressions, and |, the alternation operator, separates alternatives.

For example, D, the set of the ten single digits, is defined as

\[ D = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9). \]

The characters (, ), ', *, +, and | are meta-characters (punctuation and regular expression operators).

Meta-characters must be quoted when used as ordinary characters to avoid ambiguity.

Kleene Closure

A useful operation is Kleene closure represented by a postfix * operator.

Let P be a set of strings. Then \( P^* \) represents all strings formed by the catenation of zero or more selections (possibly repeated) from P.

Zero selections are denoted by \( \lambda \).

For example, LC* is the set of all words composed of lower-case letters, of any length (including the zero length word, \( \lambda \)).

Precisely stated, a string \( s \in P^* \) if and only if s can be broken into zero or more pieces: \( s = s_1 s_2 \ldots s_n \) so that each \( s_i \in P (n \geq 0, 1 \leq i \leq n). \)

We allow \( n = 0 \), so \( \lambda \) is always in P.

Definition of Regular Expressions

Using catenations, alternation and Kleene closure, we can define regular expressions as follows:

- \( \emptyset \) is a regular expression denoting the empty set (the set containing no strings). \( \emptyset \) is rarely used, but is included for completeness.
- \( \lambda \) is a regular expression denoting the set that contains only the empty string. This set is not the same as the empty set, because it contains one element.
- A string \( s \) is a regular expression denoting a set containing the single string \( s \).
If A and B are regular expressions, then A | B, AB, and A* are also regular expressions, denoting the alternation, catenation, and Kleene closure of the corresponding regular sets.

Each regular expression denotes a set of strings (a regular set). Any finite set of strings can be represented by a regular expression of the form \((s_1 \mid s_2 \mid \ldots \mid s_k)\). Thus the reserved words of ANSI C can be defined as (auto | break | case | ...).

The following additional operations useful. They are not strictly necessary, because their effect can be obtained using alternation, catenation, Kleene closure:

- \(P^+\) denotes all strings consisting of one or more strings in P catenated together:
  \[ P^+ = (P^+ \mid \lambda) \text{ and } P^+ = P P^*. \]
  For example, \((0 \mid 1)^+\) is the set of all strings containing one or more bits.

- If A is a set of characters, Not(A) denotes \((\Sigma - A)\); that is, all characters in \(\Sigma\) not included in A. Since Not(A) can never be larger than \(\Sigma\) and \(\Sigma\) is finite, Not(A) must also be finite, and is therefore regular. Not(A) does not contain \(\lambda\) since \(\lambda\) is not a character (it is a zero-length string).

- If \(k\) is a constant, the set \(A^k\) represents all strings formed by catenating \(k\) (possibly different) strings from A. That is, \(A^k = (A A A \ldots)\) (k copies). Thus \((0 \mid 1)^{32}\) is the set of all bit strings exactly 32 bits long.

For example, Not(Eol) is the set of all characters excluding Eol (the end of line character, '\n' in Java or C).

- It is possible to extend Not to strings, rather than just \(\Sigma\). That is, if S is a set of strings, we define S to be \((\Sigma^* - S)\); the set of all strings except those in S. Though S is usually infinite, it is also regular if S is.

- If \(k\) is a constant, the set \(A^k\) represents all strings formed by catenating \(k\) (possibly different) strings from A. Thus \((0 \mid 1)^3\) is the set of all bit strings exactly 32 bits long.

Examples

Let D be the ten single digits and let L be the set of all 52 letters. Then

- A Java or C++ single-line comment that begins with // and ends with Eol can be defined as:
  \[ \text{Comment} = // \text{Not(Eol)}^* \text{Eol} \]

- A fixed decimal literal (e.g., 12.345) can be defined as:
  \[ \text{Lit} = D^+. D^+ \]

- An optionally signed integer literal can be defined as:
  \[ \text{IntLiteral} = ( '+ ' | - | \lambda ) D^+ \]

(Why the quotes on the plus?)
A comment delimited by ## markers, which allows single #’s within the comment body:

```
Comment2 =
  ## ((# | λ) Not(#) ) * ##
```

All finite sets and many infinite sets are regular. But not all infinite sets are regular. Consider the set of balanced brackets of the form `[[[...

This set is defined formally as `{[m]^m | m ≥ 1}`.

This set is known not to be regular. Any regular expression that tries to define it either does not get all balanced nestings or it includes extra, unwanted strings.