Custom-EBook

Written by Jainish Parmar

Table of Contents

5.6 Further Reading

As its name suggests, YACC was not the first compiler construction tool, but it remains widely used today and has led to a proliferation of similar tools written in various languages and addressing different classes of grammars. Here is just a small selection:

1. S. C. Johnson, "YACC: Yet Another Compiler-Compiler", Bell Laboratories Technical Journal, 1975.
2. D. Grune and C.J.H Jacobs, "A programmer-friendly LL(1) parser generator", Software: Practice and Experience, volume 18, number 1.
3. T.J. Parr and R.W. Quong, "ANTLR: A predicated LL(k) Parser Generator", Software: Practice and Experience, 1995.
4. S. McPeak, G.C. Necula, "Elkhound: A Fast, Practical GLR Parser Generator", International Conference on Compiler Construction, 2004.

Chapter 6 – The Abstract Syntax Tree

6.1 Overview

The **abstract syntax tree (AST)** is an important internal data structure that represents the primary structure of a program. The AST is the starting point for semantic analysis of a program. It is "abstract" in the sense that the structure leaves out the particular details of parsing: the AST does not care whether a language has prefix, postfix, or infix expressions. (In fact, the AST we describe here can be used to represent most procedural languages.)

For our project compiler, we will define an AST in terms of five C structures representing declarations, statements, expressions, types, and parameters. While you have certainly encountered each of these terms while learning programming, they are not always used precisely in practice. This chapter will help you to sort those items out very clearly:

- A **declaration** states the name, type, and value of a symbol so that it can be used in the program. Symbols include items such as constants, variables, and functions.
- A **statement** indicates an action to be carried out that changes the state of the program. Examples include loops, conditionals, and function returns.
- An **expression** is a combination of values and operations that is **evaluated** according to specific rules and yields a **value** such as an integer, floating point, or string. In some programming languages, an expression may also have a **side effect** that changes the state of the program.

For each kind of element in the AST, we will give an example of the code and how it is constructed. Because each of these structures potentially has pointers to each of the other types, it is necessary to preview all of them before seeing how they work together.

Once you understand all of the elements of the AST, we finish the chapter by demonstrating how the entire structure can be created automatically through the use of the Bison parser generator.

6.2 Declarations

A complete B-Minor program is a sequence of declarations. Each declaration states the existence of a variable or a function. A variable declaration may optionally give an initializing value. If none is given, it is given a default value of zero. A function declaration may optionally give the body of the function in code; if no body is given, then the declaration serves as a prototype for a function declared elsewhere.

For example, the following are all valid declarations:

```
b: boolean;
s: string = "hello";
f: function integer ( x: integer ) = { return x*x; }
```

A declaration is represented by a `decl` structure that gives the name, type, value (if an expression), code (if a function), and a pointer to the next declaration in the program:

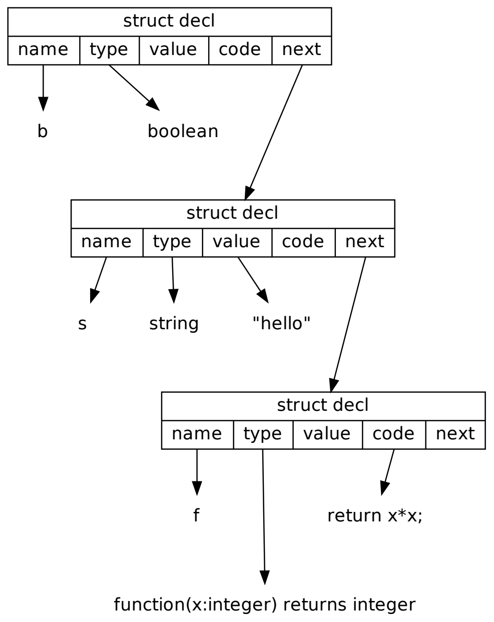
```
struct decl {
    char *name;
    struct type *type;
    struct expr *value;
    struct stmt *code;
    struct decl *next;
};
```

Because we will be creating a lot of these structures, you will need a factory function that allocates a structure and initializes its fields, like this:

```
struct decl * decl_create( char *name,
                           struct type *type,
                           struct expr *value,
                           struct stmt *code,
                           struct decl *next )
{
    struct decl *d = malloc(sizeof(*d));
    d->name = name;
    d->type = type;
    d->value = value;
    d->code = code;
    d->next = next;
    return d;
}
```

(You will need to write similar code for statements, expressions, etc, but we won't keep repeating it here.)

The three declarations on the preceding page can be represented graphically as a linked list, like this:



Note that some of the fields point to nothing: these would be represented by a null pointer, which we omit for clarity. Also, our picture is incomplete and must be expanded: the items representing types, expressions, and statements are all complex structures themselves that we must describe.

6.3 Statements

The body of a function consists of a sequence of statements. A statement indicates that the program is to take a particular action in the order specified, such as computing a value, performing a loop, or choosing between branches of an alternative. A statement can also be a declaration of a local variable. Here is the `stmt` structure:

```
struct stmt {                        typedef enum {
    stmt_t kind;                         STMT_DECL,
    struct decl *decl;                   STMT_EXPR,
    struct expr *init_expr;              STMT_IF_ELSE,
    struct expr *expr;                   STMT_FOR,
    struct expr *next_expr;              STMT_PRINT,
    struct stmt *body;                   STMT_RETURN,
    struct stmt *else_body;              STMT_BLOCK
    struct stmt *next;               } stmt_t;
};
```

The `kind` field indicates what kind of statement it is:

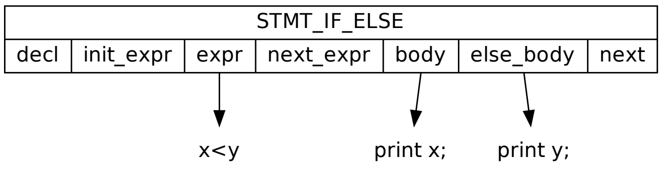
- `STMT_DECL` indicates a (local) declaration, and the `decl` field will point to it.
- `STMT_EXPR` indicates an expression statement and the `expr` field will point to it.
- `STMT_IF_ELSE` indicates an if-else expression such that the `expr` field will point to the control expression, the `body` field to the statements executed if it is true, and the `else_body` field to the statements executed if it is false.
- `STMT_FOR` indicates a for-loop, such that `init_expr`, `expr`, and `next_expr` are the three expressions in the loop header, and `body` points to the statements in the loop.
- `STMT_PRINT` indicates a `print` statement, and `expr` points to the expressions to print.
- `STMT_RETURN` indicates a `return` statement, and `expr` points to the expression to return.
- `STMT_BLOCK` indicates a block of statements inside curly braces, and `body` points to the contained statements.

And, as we did with declarations, we require a function `stmt_create` to create and return a statement structure:

```
struct stmt * stmt_create( stmt_t kind,
    struct decl *decl, struct expr *init_expr,
    struct expr *expr, struct expr *next_expr,
    struct stmt *body, struct stmt *else_body,
    struct stmt *next );
```

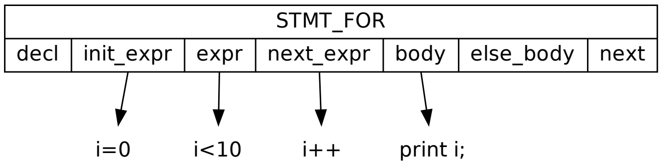
This structure has a lot of fields, but each one serves a purpose and is used when necessary for a particular kind of statement. For example, an if-else statement only uses the `expr`, `body`, and `else_body` fields, leaving the rest null:

```
if( x<y ) print x; else print y;
```



A for-loop uses the three `expr` fields to represent the three parts of the loop control, and the `body` field to represent the code being executed:

```
for(i=0;i<10;i++) print i;
```



6.4 Expressions

Expressions are implemented much like the simple expression AST shown in Chapter 5. The difference is that we need many more binary types: one for every operator in the language, including arithmetic, logical, comparison, assignment, and so forth. We also need one for every type of leaf value, including variable names, constant values, and so forth. The `name` field will be set for `EXPR_NAME`, the `integer_value` field for `EXPR_INTEGER_LITERAL`, and so on. You may need to add values and types to this structure as you expand your compiler.

```
struct expr {              typedef enum {
    expr_t kind;               EXPR_ADD,
    struct expr *left;         EXPR_SUB,
    struct expr *right;        EXPR_MUL,
                               EXPR_DIV,
    const char *name;          ...
    int integer_value;         EXPR_NAME,
    const char *               EXPR_INTEGER_LITERAL,
       string_literal;         EXPR_STRING_LITERAL
};                         } expr_t;
```

As before, you should create a factory for a binary operator:

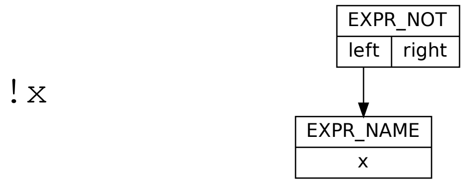
```
struct expr * expr_create( expr_t kind,
                           struct expr *L,
                           struct expr *R );
```

And then a factory for each of the leaf types:

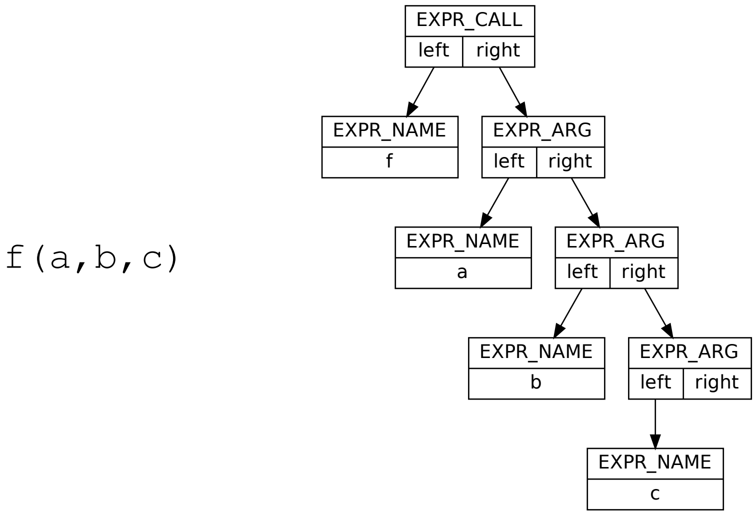
```
struct expr * expr_create_name( const char *name );
struct expr * expr_create_integer_literal( int i );
struct expr * expr_create_boolean_literal( int b );
struct expr * expr_create_char_literal( char c );
struct expr * expr_create_string_literal
                               ( const char *str );
```

Note that you can store the integer, boolean, and character literal values all in the `integer_value` field.

A few cases deserve special mention. Unary operators like logical-not typically have their sole argument in the `left` pointer:

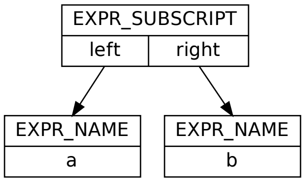


A function call is constructed by creating an `EXPR_CALL` node, such that the left-hand side is the function name, and the right hand side is an unbalanced tree of `EXPR_ARG` nodes. While this looks a bit awkward, it allows us to express a linked list using a tree, and will simplify the handling of function call arguments on the stack during code generation.



Array subscripting is treated like a binary operator, such that the name of the array is on the left side of the EXPR_SUBSCRIPT operator, and an integer expression on the right:

```
a[b]
```



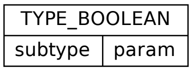
6.5 Types

A type structure encodes the type of every variable and function mentioned in a declaration. Primitive types like integer and boolean are expressed by simply setting the kind field appropriately, and leaving the other fields null. Compound types like array and function are built by connecting multiple type structures together.

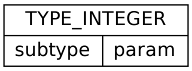
```
typedef enum {
    TYPE_VOID,
    TYPE_BOOLEAN,
    TYPE_CHARACTER,
    TYPE_INTEGER,
    TYPE_STRING,
    TYPE_ARRAY,
    TYPE_FUNCTION
} type_t;

struct type {
    type_t kind;
    struct type *subtype;
    struct param_list *params;
};

struct param_list {
    char *name;
    struct type *type;
    struct param_list *next;
};
```

For example, to express a basic type like a boolean or an integer, we simply create a standalone `type` structure, with `kind` set appropriately, and the other fields null:

```
boolean
```

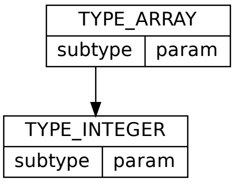


```
integer
```



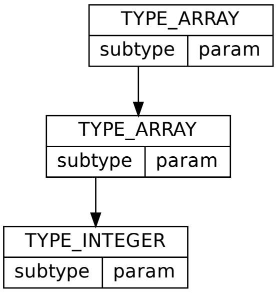
To express a compound type like an array of integers, we set `kind` to `TYPE_ARRAY` and set `subtype` to point to a `TYPE_INTEGER`:

```
array [] integer
```



These can be linked to arbitrary depth, so to express an array of array of integers:

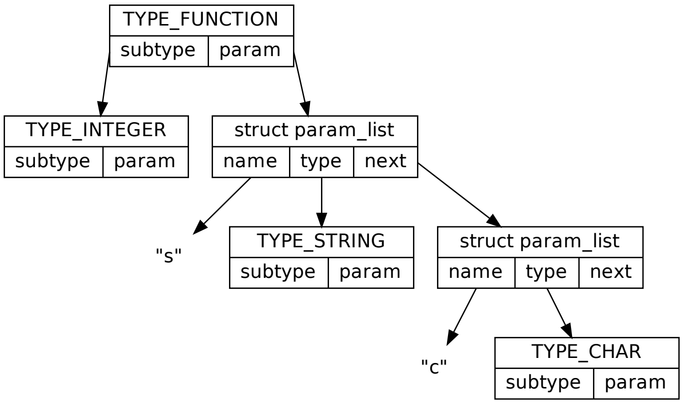
```
array [] array [] integer
```



To express the type of a function, we use `subtype` to express the return type of the function, and then connect a linked list of `param_list` nodes to describe the name and type of each parameter to the function.

For example, here is the type of a function which takes two arguments and returns an integer:

```
function integer (s:string, c:char)
```



Note that the type structures here let us express some complicated and powerful higher order concepts of programming. By simply swapping in complex types, you can describe an array of ten functions, each returning an integer:

```
a: array [10] function integer ( x: integer );
```

Or how about a function that returns a function?

```
f: function function integer (x:integer) (y:integer);
```

Or even a function that returns an array of functions!

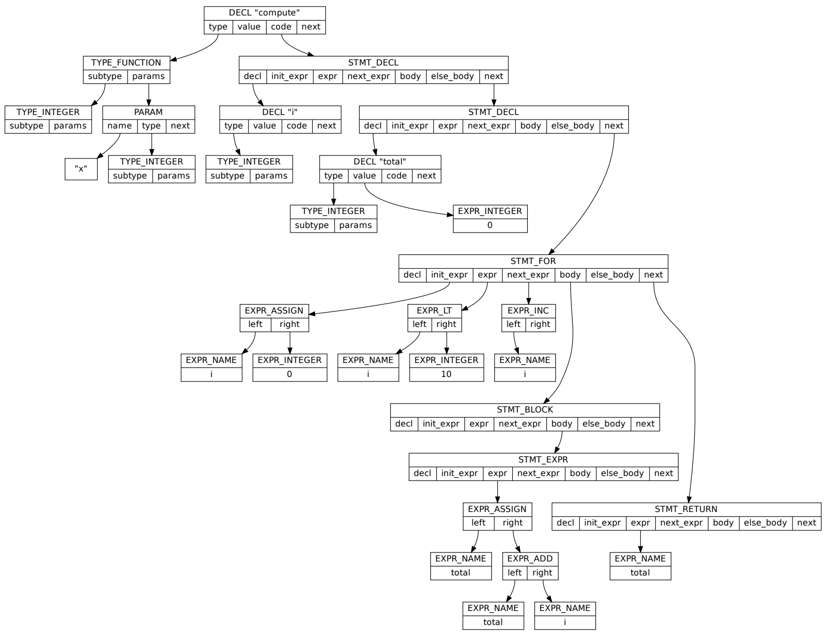
```
g: function array [10]
        function integer (x:integer) (y:integer);
```

While the B-Minor type system is capable of *expressing* these ideas, these combinations will be rejected later in typechecking, because they require a more dynamic implementation than we are prepared to create. If you find these ideas interesting, then you should read up on functional languages such as Scheme and Haskell.

6.6 Putting it All Together

Now that you have seen each individual component, let's see how a complete B-Minor function would be expressed as an AST:

```
compute: function integer ( x:integer ) = {
        i: integer;
        total: integer = 0;
        for(i=0;i<10;i++) {
                total = total + i;
        }
        return total;
}
```



6.7 Building the AST

With the functions created so far in this chapter, we could, in principle, construct the AST manually in a sort of nested style. For example, the following code represents a function called `square` which accepts an integer `x` as a parameter, and returns the value `x*x`:

```
d = decl_create(
   "square",
   type_create(TYPE_FUNCTION,
       type_create(TYPE_INTEGER,0,0),
       param_list_create(
           "x",
           type_create(TYPE_INTEGER,0,0),
           0)),
   0,
   stmt_create(STMT_RETURN,0,0,
       expr_create(EXPR_MUL,
           expr_create_name("x"),
           expr_create_name("x")),
       0,0,0,0),
   0);
```

Obviously, this is no way to write code! Instead, we want our parser to invoke the various creation functions whenever they are reduced, and then hook them up into a complete tree. Using an LR parser generator like Bison, this process is straightforward. (Here I will give you the idea of how to proceed, but you will need to figure out many of the details in order to complete the parser.)

At the top level, a B-Minor program is a sequence of declarations:

```
program : decl_list
          { parser_result = $1; }
        ;
```

Then, we write rules for each of the various kinds of declarations in a B-Minor program:

```
decl : name TOKEN_COLON type TOKEN_SEMI
     { $$ = decl_create($1,$3,0,0,0); }
   | name TOKEN_COLON type TOKEN_ASSIGN expr TOKEN_SEMI
     { $$ = decl_create($1,$3,$5,0,0); }
   | /* and more cases here */
       . . .
   ;
```

Since each `decl` structure is created separately, we must connect them together in a linked list formed by a `decl_list`. This is most easily done by making the rule right-recursive, so that `decl` on the left represents one declaration, and `decl_list` on the right represents the remainder of the linked list. The end of the list is a null value when `decl_list` produces ϵ.

```
decl_list : decl decl_list
            { $$ = $1; $1->next = $2; }
          | /* epsilon */
            { $$ = 0; }
          ;
```

For each kind of statement, we create a `stmt` structure that pulls out the necessary elements from the grammar.

```
stmt : TOKEN_IF TOKEN_LPAREN expr TOKEN_RPAREN stmt
       { $$ = stmt_create(STMT_IF_ELSE,0,0,$3,0,$5,0,0); }
     | TOKEN_LBRACE stmt_list TOKEN_RBRACE
       { $$ = stmt_create(STMT_BLOCK,0,0,0,0,$2,0,0); }
     | /* and more cases here */
       . . .
     ;
```

Proceed in this way down through each of the grammar elements of a B-Minor program: declarations, statements, expressions, types, parameters, until you reach the leaf elements of literal values and symbols, which are handled in the same way as in Chapter 5.

There is one last complication: What, exactly is the semantic type of the values returned as each rule is reduced? It isn't a single type, because each kind of rule returns a different data structure: a declaration rule returns a `struct decl *`, while an identifier rule returns a `char *`. To make this work, we inform Bison that the semantic value is the union of all of the types in the AST:

```
%union {
        struct decl *decl;
        struct stmt *stmt;
        . . .
        char *name;
};
```

And then indicate the specific subfield of the union used by each rule:

```
%type <decl> program decl_list decl . . .
%type <stmt> stmt_list stmt . . .
. . .
%type <name> name
```

6.8 Exercises

1. Write a complete LR grammar for B-Minor and test it using Bison. Your first attempt will certainly have many shift-reduce and reduce-reduce conflicts, so use your knowledge of grammars from Chapter 4 to rewrite the grammar and eliminate the conflicts.

2. Write the AST structures and generating functions as outlined in this chapter, and manually construct some simple ASTs using nested function calls as shown above.

3. Add new functions `decl_print()`, `stmt_print()`, etc. that print the AST back out so you can verify that the program was generated correctly. Make your output nicely formatted using indentation and consistent spacing, so that the code is easily readable.

4. Add the AST generator functions as action rules to your Bison grammar, so that you can parse complete programs, and print them back out again.

5. Add new functions `decl_translate()`, `stmt_translate()`, etc. that output the B-Minor AST in a different language of your own choosing, such as Python or Java or Rust.

6. Add new functions that emit the AST in a graphical form so you can “see” the structure of a program. One approach would be to use the Graphviz DOT format: let each declaration, statement, etc be a node in a graph, and then let each pointer between structures be an edge in the graph.

Chapter 7 – Semantic Analysis

Now that we have completed construction of the AST, we are ready to begin analyzing the **semantics**, or the actual meaning of a program, and not simply its structure.

Type checking is a major component of semantic analysis. Broadly speaking, the type system of a programming language gives the programmer a way to make verifiable assertions that the compiler can check automatically. This allows for the detection of errors at compile-time, instead of at runtime.

Different programming languages have different approaches to type checking. Some languages (like C) have a rather weak type system, so it is possible to make serious errors if you are not careful. Other languages (like Ada) have very strong type systems, but this makes it more difficult to write a program that will compile at all!

Before we can perform type checking, we must determine the type of each identifier used in an expression. However, the mapping between variable names and their actual storage locations is not immediately obvious. A variable `x` in an expression could refer to a local variable, a function parameter, a global variable, or something else entirely. We solve this problem by performing **name resolution**, in which each definition of a variable is entered into a **symbol table**. This table is referenced throughout the semantic analysis stage whenever we need to evaluate the correctness of some code.

Once name resolution is completed, we have all the information necessary to check types. In this stage, we compute the type of complex expressions by combining the basic types of each value according to standard conversion rules. If a type is used in a way that is not permitted, the compiler will output an (ideally helpful) error message that will assist the programmer in resolving the problem.

Semantic analysis also includes other forms of checking the correctness of a program, such as examining the limits of arrays, avoiding bad pointer traversals, and examining control flow. Depending on the design of the language, some of these problems can be detected at compile time, while others may need to wait until runtime.

7.1 Overview of Type Systems

Most programming languages assign to every value (whether a literal, constant, or variable) a **type**, which describes the interpretation of the data in that variable. The type indicates whether the value is an integer, a floating point number, a boolean, a string, a pointer, or something else. In most languages, these atomic types can be combined into higher-order types such as enumerations, structures, and variant types to express complex constraints.

The type system of a language serves several purposes:

- **Correctness.** A compiler uses type information provided by the programmer to raise warnings or errors if a program attempts to do something improper. For example, it is almost certainly an error to assign an integer value to a pointer variable, even though both might be implemented as a single word in memory. A good type system can help to eliminate runtime errors by flagging them at compile time instead.

- **Performance.** A compiler can use type information to find the most efficient implementation of a piece of code. For example, if the programmer tells the compiler that a given variable is a constant, then the same value can be loaded into a register and used many times, rather than constantly loading it from memory.

- **Expressiveness.** A program can be made more compact and expressive if the language allows the programmer to leave out facts that can be inferred from the type system. For example, in B-Minor the `print` statement does not need to be told whether it is printing an integer, a string, or a boolean: the type is inferred from the expression and the value is automatically displayed in the proper way.

A programming language (and its type system) are commonly classified on the following axes:

- safe or unsafe
- static or dynamic
- explicit or implicit

In an **unsafe programming language**, it is possible to write valid programs that have wildly undefined behavior that violates the basic structure of the program. For example, in the C programming language, a program can construct an arbitrary pointer to modify any word in memory, and thereby change the data and code of the compiled program. Such power is probably necessary to implement low-level code like an operating system or a driver, but is problematic in general application code.

For example, the following code in C is syntactically legal and will compile, but is unsafe because it writes data outside the bounds of the array `a[]`. As a result, the program could have almost any outcome, including incorrect output, silent data corruption, or an infinite loop.

```
/* This is C code */
int i;
int a[10];
for(i=0;i<100;i++) a[i] = i;
```

In a **safe programming language**, it is not possible to write a program that violates the basic structures of the language. That is, no matter what input is given to a program written in a safe language, it will always execute in a well-defined way that preserves the abstractions of the language. A safe programming language enforces the boundaries of arrays, the use of pointers, and the assignment of types to prevent undefined behavior. Most interpreted languages, like Perl, Python, and Java, are safe languages.

For example, in C#, the boundaries of arrays are checked at runtime, so that running off the end of an array has the predictable effect of throwing an `IndexOutOfRangeException`:

```
/* This is C-sharp code */
a = new int[10];
for(int i=0;i<100;i++) a[i] = i;
```

In a **statically typed language**, all typechecking is performed at compile-time, long before the program runs. This means that the program can be translated into basic machine code without retaining any of the type information, because all operations have been checked and determined to be safe. This yields the most high performance code, but it does eliminate some kinds of convenient programming idioms.

Static typing is often used to distinguish between integer and floating point operations. While operations like addition and multiplication are usually represented by the same symbols in the source language, they are implemented with fundamentally different machine code. For example, in the C language on X86 machines, `(a+b)` would be translated to an `ADDL` instruction for integers, but an `FADD` instruction for floating point values. To know which instruction to apply, we must first determine the type of `a` and `b` and deduce the intended meaning of `+`.

In a **dynamically typed language**, type information is available at runtime and stored in memory alongside the data that it describes. As the program executes, the safety of each operation is checked by comparing the types of each operand. If types are observed to be incompatible, then the program must halt with a runtime type error. This also allows for

2.2.4 Corollary [Principle of lexicographic induction]: Define the following "dictionary ordering" on pairs of natural numbers: $(m, n) < (m', n')$ iff $m < m'$ or $m = m'$ and $n < n'$.

Now, suppose that P is some predicate on pairs of natural numbers. If we can show, for each (m, n), that $(\forall(m', n') < (m, n).\ P(m', n'))$ implies $P(m, n)$, then we may conclude that $P(m, n)$ holds for every pair (m, n).

(A similar principle holds for lexicographically ordered triples, quadruples, etc.) □

Proof: The lexicographic ordering on pairs of numbers is well founded. □

2.3 Term Rewriting

(or maybe this material should be folded into the next chapter...)

Chapter 3

Untyped Arithmetic Expressions

Quite a bit of text and a few technical definitions are still missing here. The idea is to introduce the basic ideas of defining a language and its operational semantics formally, and proving some simple properties by structural induction, before getting to the complexities (esp. name binding) of the full-blown lambda-calculus.

3.1 Basics

We begin with a very simple language for calculating with numbers and booleans.

```
  true;
▶ true

  if false then true else false;
▶ false

  0;
▶ 0

  succ (succ (succ 0));
▶ 3

  succ (pred 0);
▶ 1

  iszero (pred (succ 0));
```

```
▸ true

  iszero (succ (succ 0));

▸ false
```

Throughout the book, the symbol ▸ will be used to display the results of evaluating examples. You can think of the lines marked with ▸ as the responses from an interactive interpreter when presented with the preceding inputs.

For brevity, the examples use standard arabic numerals as shorthand for nested applications of `succ` to `0`, writing `succ(succ(succ(0)))` as `3`.

Syntax

The syntax of arithmetic expressions comprises several kinds of **terms**. The constants `true`, `false`, and `0` are terms. If `t` is a term, then so are `succ t`, `pred t`, and `iszero t`. Finally, if t_1, t_2, and t_3 are terms, then so is `if` t_1 `then` t_2 `else` t_3. These forms are summarized in the following **abstract grammar**:

`t ::=`		(*terms...*)
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*

Evaluation

3.2 Formalities

Syntax

3.2.1 Definition [Terms]: The set of terms is the smallest set $\mathcal{T}$ such that

1. $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$;
2. if $t_1 \in \mathcal{T}$, then $\{\texttt{succ } t_1, \texttt{pred } t_1, \texttt{iszero } t_1\} \subseteq \mathcal{T}$;
3. if $t_1 \in \mathcal{T}$, $t_2 \in \mathcal{T}$, and $t_3 \in \mathcal{T}$, then $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \in \mathcal{T}$.

These three clauses capture exactly what is meant by the productions in the more concise and readable "abstract grammar" notation that we used above. □

Definition 3.2.1 is an example of an **inductive definition**. Since inductive definitions are ubiquitous in the study of programming languages, it is worth pausing for a moment to examine this one in detail. Here is an alternative definition of the same set, in a more concrete style.

3.2.2 Definition [Terms, more concretely]: For each natural number i, define a set $\mathcal{S}_i$ as follows:

$$\begin{array}{lcl} \mathcal{S}_0 & = & \emptyset \\ \mathcal{S}_{i+1} & = & \quad \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ & & \cup \ \{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1 \mid \texttt{t}_1 \in \mathcal{S}_i\} \\ & & \cup \ \{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in \mathcal{S}_i\}. \end{array}$$

Finally, let

$$\mathcal{S} \quad = \quad \bigcup_i \mathcal{S}_i.$$

That is, $\mathcal{S}_0$ is empty; $\mathcal{S}_1$ contains just the constants; $\mathcal{T}_2$ contains the constants plus the phrases that can be built with constants and just one `succ`, `pred`, `iszero`, or `if`; $\mathcal{S}_3$ contains these plus all phrases that can be built using `succ`, `pred`, `iszero`, and `if` on phrases in $\mathcal{S}_2$; and so on. $\mathcal{S}$ collects together all the phrases that can be built in this way—i.e., all phrases built by some finite number of applications and abstractions, beginning with just variables. □

3.2.3 Exercise [Quick check]: List the elements of $\mathcal{S}_3$. □

3.2.4 Exercise: Show that the sets $\mathcal{S}_i$ are **cumulative**—that is, that for each i we have $\mathcal{S}_i \subseteq \mathcal{S}_{i+1}$. □

Now let us check that the two definitions of terms actually define the same set. We'll do the proof in quite a bit of detail, to show how all the pieces fit together.

3.2.5 Proposition: $\mathcal{T} = \mathcal{S}$. □

Proof: $\mathcal{T}$ was defined as the smallest set satisfying certain conditions. So it suffices to show (a) that $\mathcal{S}$ satisfies these conditions, and (b) that any set satisfying the conditions has $\mathcal{S}$ as a subset (i.e., that $\mathcal{S}$ is the *smallest* set satisfying the conditions).

For part (a), we must check that each of the three conditions in Definition 3.2.1 holds of $\mathcal{S}$. First, since $\mathcal{S}_1 = \{\texttt{true}, \texttt{false}, \texttt{0}\}$ and $\mathcal{S}_1 \subseteq \bigcup_i \mathcal{S}_i$, it is clear that the constants are in $\mathcal{S}$. Second, if $\texttt{t}_1 \in \mathcal{S}$, then (since $\mathcal{S} = \bigcup_i \mathcal{S}_i$) there must be some i such that $\texttt{t}_i \in \mathcal{S}_i$. But then, by the definition of $\mathcal{S}_{i+1}$, we must have $\texttt{succ t}_1 \in \mathcal{S}_{i+1}$, hence $\texttt{succ t}_1 \in \mathcal{S}$; similarly, we see that $\texttt{pred t}_1 \in \mathcal{S}$ and $\texttt{iszero t}_1 \in \mathcal{S}$. Third, if $\texttt{t}_1 \in \mathcal{S}$, $\texttt{t}_2 \in \mathcal{S}$, and $\texttt{t}_3 \in \mathcal{S}$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{S}$, by a similar argument.

For part (b), suppose that some set $\mathcal{S}'$ satisfies the three conditions in Definition 3.2.1. We will argue, by induction on n, that every $\mathcal{S}_n \subseteq \mathcal{S}'$, from which it will clearly follow that $\mathcal{S} \subseteq \mathcal{S}'$. Suppose that $\mathcal{S}_m \subseteq \mathcal{S}'$ for all $m < n$; we must then show that $\mathcal{S}_n \subseteq \mathcal{S}'$. Since the definition of $\mathcal{S}_n$ has two clauses (for $n = 0$ and $n = i + 1$), there are two cases to consider.

- If $n = 0$, then $\mathcal{S}_n = \emptyset$. But $\emptyset \subseteq \mathcal{S}'$ trivially.
- Otherwise, $n = i + 1$ for some i. Let `t` be some element of $\mathcal{S}_{i+1}$. Since $\mathcal{S}_{i+1}$ is defined as the union of three smaller sets, `t` must come from one of these sets, and there are three possibilities to consider:
  1. `t` is a constant, hence $\mathtt{t} \in \mathcal{S}'$ by condition (1).
  2. `t` has the form `succ` $\mathtt{t}_1$, `pred` $\mathtt{t}_1$, or `iszero` $\mathtt{t}_1$, for some $\mathtt{t}_1 \in \mathcal{S}_i$. But then, by the induction hypothesis, $\mathtt{t}_1 \in \mathcal{S}'$, and so, by condition (2), $\mathtt{t} \in \mathcal{S}'$.
  3. `t` has the form `if` $\mathtt{t}_1$ `then` $\mathtt{t}_2$ `else` $\mathtt{t}_3$, for some $\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3 \in \mathcal{S}_i$. Again, by the induction hypothesis, $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$ are all in $\mathcal{S}'$, and hence, by condition (3), so is `t`.

Thus, we have shown that each $\mathcal{S}_i \subseteq \mathcal{S}'$. By the definition of $\mathcal{S}$ as the union of all the $\mathcal{S}_i$, this gives $\mathcal{S} \subseteq \mathcal{S}'$, completing the argument. □

The explicit characterization of $\mathcal{T}$ justifies an important principle for reasoning about its elements. If $\mathtt{t} \in \mathtt{T}$, then one of three things must be true about `t`—either

1. `t` is a constant, or
2. `t` has the form `succ` $\mathtt{t}_1$, `pred` $\mathtt{t}_1$, or `iszero` $\mathtt{t}_1$ for some *smaller* term $\mathtt{t}_1$, or
3. `t` has the form `if` $\mathtt{t}_1$ `then` $\mathtt{t}_2$ `else` $\mathtt{t}_3$ for some *smaller* terms $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$.

We can put this observation to work in two ways: we can give *inductive definitions* of functions over the set of terms, and we can give *inductive proofs* of properties of terms. For example, here is a simple inductive definition of a function mapping each term `t` to the set of constants used in `t`.

3.2.6 Definition: The set of constants appearing in a term `t`, written *Consts*(`t`), is defined as follows:

$$\begin{array}{lcl}
\mathit{Consts}(\mathtt{true}) & = & \{\mathtt{true}\} \\
\mathit{Consts}(\mathtt{false}) & = & \{\mathtt{false}\} \\
\mathit{Consts}(\mathtt{iszero}) & = & \{\mathtt{iszero}\} \\
\mathit{Consts}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_1)
\end{array}$$ □

Another property of terms that can be calculated by an inductive defintion is their size.

3.2.7 Definition: The **size** of a term t, written *size*(t), is defined as follows:

$$\begin{array}{lcl}
\mathit{size}(\texttt{true}) & = & 1 \\
\mathit{size}(\texttt{false}) & = & 1 \\
\mathit{size}(\texttt{iszero}) & = & 1 \\
\mathit{size}(\texttt{succ t}_1) & = & \mathit{size}(\texttt{t}_1) + 1 \\
\mathit{size}(\texttt{pred t}_1) & = & \mathit{size}(\texttt{t}_1) + 1 \\
\mathit{size}(\texttt{iszero t}_1) & = & \mathit{size}(\texttt{t}_1) + 1 \\
\mathit{size}(\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3) & = & \mathit{size}(\texttt{t}_1) + \mathit{size}(\texttt{t}_1) + \mathit{size}(\texttt{t}_1) + 1
\end{array}$$

That is, the size of t is the number of nodes in its **abstract syntax tree**. Similarly, the **depth** of a term t, written *depth*(t), is defined as follows:

$$\begin{array}{lcl}
\mathit{depth}(\texttt{true}) & = & 1 \\
\mathit{depth}(\texttt{false}) & = & 1 \\
\mathit{depth}(\texttt{iszero}) & = & 1 \\
\mathit{depth}(\texttt{succ t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{pred t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{iszero t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3) & = & \max(\mathit{depth}(\texttt{t}_1), \mathit{depth}(\texttt{t}_1), \mathit{depth}(\texttt{t}_1)) + 1
\end{array}$$

Equivalently, *depth*(t), is the smallest i such that $\texttt{t} \in \mathcal{S}_i$. □

Here is an inductive proof of a simple fact relating the number of constants in a term to its size.

3.2.8 Lemma: The number of distinct constants in a term t is always smaller than the size of t ($|\mathit{Consts}(\texttt{t})| \leq \mathit{size}(\texttt{t})$). □

Proof: The property in itself is entirely obvious, of course. What's interesting is the form of the inductive proof, which we'll see repeated many times as we go along.

The proof proceeds by induction on the size of t. That is, assuming the desired property for all terms smaller than t, we must prove it for t itself; if we can do this, we may conclude that the property holds for all t. There are three cases to consider:

Case: t is a constant

Immediate: $|\mathit{Consts}(\texttt{t})| = |\{\texttt{t}\}| = 1 = \mathit{size}(\texttt{t})$.

Case: $\texttt{t} = \texttt{succ t}_1, \texttt{pred t}_1, \texttt{oriszero t}_1$

By the induction hypothesis, $|\mathit{Consts}(\texttt{t}_1)| \leq \mathit{size}(\texttt{t}_1)$. We now calculate as follows: $|\mathit{Consts}(\texttt{t})| = |\mathit{Consts}(\texttt{t}_1)| \leq \mathit{size}(\texttt{t}_1) < \mathit{size}(\texttt{t})$.

Case: $\texttt{t} = \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$

By the induction hypothesis, $|\mathit{Consts}(\texttt{t}_1)| \leq \mathit{size}(\texttt{t}_1)$ and $|\mathit{Consts}(\texttt{t}_2)| \leq \mathit{size}(\texttt{t}_2)$ and $|\mathit{Consts}(\texttt{t}_3)| \leq \mathit{size}(\texttt{t}_3)$. We now calculate as follows: $|\mathit{Consts}(\texttt{t})| = |\mathit{Consts}(\texttt{t}_1) \cup \mathit{Consts}(\texttt{t}_2) \cup \mathit{Consts}(\texttt{t}_3)| \leq |\mathit{Consts}(\texttt{t}_1)| + |\mathit{Consts}(\texttt{t}_2)| + |\mathit{Consts}(\texttt{t}_3)| \leq \mathit{size}(\texttt{t}_1) + \mathit{size}(\texttt{t}_2) + \mathit{size}(\texttt{t}_3) < \mathit{size}(\texttt{t})$. □

The form of this proof can be clarified by restating it as a general reasoning principle. (Compare this principle with the induction principle for natural numbers on p. 18.)

3.2.9 Theorem [Principle of induction on terms]: Suppose that P is some predicate on terms. If we can show, for each s, that $(\forall \texttt{r}.\ \mathit{size}(\texttt{r}) < \mathit{size}(\texttt{s}) \text{ implies } \mathsf{P}(\texttt{r}))$ implies $\mathsf{P}(\texttt{s})$, then we may conclude that $\mathsf{P}(\texttt{t})$ holds for every term t. □

Proof: Exercise. □

Evaluation

3.2.10 Definition: The set of **values** is the subset of terms defined by the following abstract grammar:

v ::=		*(values...)*
	`true`	*value true*
	`false`	*value false*
	`0`	*zero value*
	`succ v`	*successor value*

3.2.11 Definition: The **one-step evaluation** relation $\longrightarrow$ is the smallest relation containing all instances of the following rules:

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2 \qquad \text{(E-BOOLBETAT)}$$

$$\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \qquad \text{(E-BOOLBETAF)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \qquad \text{(E-IF)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{succ } \texttt{t}_1 \longrightarrow \texttt{succ } \texttt{t}_1'} \qquad \text{(E-SUCC)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-BETANATPZ)}$$

$$\texttt{pred (succ v)} \longrightarrow \texttt{v} \qquad \text{(E-BETANATPS)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{pred } \texttt{t}_1 \longrightarrow \texttt{pred } \texttt{t}_1'} \qquad \text{(E-PRED)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-BetaNatIZ)}$$

$$\texttt{iszero (succ v)} \longrightarrow \texttt{false} \qquad \text{(E-BetaNatIS)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{iszero t}_1 \longrightarrow \texttt{iszero t}_1'} \qquad \text{(E-IsZero)}$$

□

3.2.12 Definition: The **multi-step evaluation** relation $\longrightarrow^*$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that

- if $\texttt{t} \longrightarrow \texttt{t}'$ then $\texttt{t} \longrightarrow^* \texttt{t}'$,
- $\texttt{t} \longrightarrow^* \texttt{t}$ for all $\texttt{t}$, and
- if $\texttt{t} \longrightarrow^* \texttt{t}'$ and $\texttt{t}' \longrightarrow^* \texttt{t}''$, then $\texttt{t} \longrightarrow^* \texttt{t}''$. □

3.2.13 Exercise [Quick check]: Rewrite the previous definition using a set of inference rules to define the relation $\texttt{t} \longrightarrow^* \texttt{t}'$. (Solution on page 253.) □

3.2.14 Definition: A term $\texttt{t}$ is in **normal form** if no evaluation rule applies to it—i.e., if there is no $\texttt{t}'$ such that $\texttt{t} \longrightarrow \texttt{t}'$. □

3.2.15 Definition: An **evaluation sequence** starting from a term $\texttt{t}$ is a (finite or infinite) sequence of terms $\texttt{t}_1, \texttt{t}_2, \ldots$, such that

$\texttt{t} \longrightarrow \texttt{t}_1$
$\texttt{t}_1 \longrightarrow \texttt{t}_2$
etc. □

3.2.16 Definition: A term is said to be **stuck** if it is a normal form but not a value. □

3.2.17 Exercise: Write an abstract grammar that generates all (and only) the stuck arithmetic expressions. □

"Stuckness" gives us a simple notion of "run-time type error" for this rather abstract abstract machine. Intuitively, it characterizes the situations where the operational semantics does not know what to do because the program has reached a "meaningless state." A more serious implementation might choose other behavior in these cases, such as dumping core.

3.3 Properties

3.3.1 Proposition: Every value is in normal form. □

Proof: By inspection of the definitions of values and one-step evaluation. □

3.3.2 Proposition [Determinacy of evaluation]: If $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\mathtt{t} \longrightarrow \mathtt{t}''$, then $\mathtt{t}' = \mathtt{t}''$. □

Proof: Exercise. (Solution on page 253.) □

3.3.3 Definition: A value $\mathtt{v}$ is the **result** of a term $\mathtt{t}$ if $\mathtt{t} \longrightarrow^* \mathtt{v}$. □

3.3.4 Proposition [Uniqueness of results]: If $\mathtt{v}$ and $\mathtt{w}$ are both results of $\mathtt{t}$, then $\mathtt{v} = \mathtt{w}$. □

Proof: Exercise. □

3.4 Implementation

Explanatory text still needs to be written.

Note that this implementation is optimized for readability and for correspondence with the mathematical definitions, not for efficient execution! We'll ignore parsing and printing, the top-level read-eval-print loop, etc. Interested readers are encouraged to have a look at the ML code for the whole typechecker.

Syntax

```
type info

type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

The `info` components of this datatype are extracted, as necessary, by the error printing functions.

Evaluation

```
exception No

let rec eval1 t =
  match t with
    TmIf(fi,t1,t2,t3) when not (isval t1) →
      let t1' = eval1 t1 in
      TmIf(fi, t1', t2, t3)
  | TmIf(fi,TmTrue(_),t2,t3) →
      t2
  | TmIf(fi,TmFalse(_),t2,t3) →
      t3
  | TmSucc(fi,t1) when not (isval t1) →
      let t1' = eval1 t1 in
      TmSucc(fi, t1')
  | TmPred(fi,t1) when not (isval t1) →
      let t1' = eval1 t1 in
      TmPred(fi, t1')
  | TmPred(_,TmZero(_)) →
      TmZero(unknown)
  | TmPred(_,TmSucc(_,v1)) →
      v1
  | TmIsZero(fi,t1) when not (isval t1) →
      let t1' = eval1 t1 in
      TmIsZero(fi, t1')
  | TmIsZero(_,TmZero(_)) →
      TmTrue(unknown)
  | TmIsZero(_,TmSucc(_,_)) →
      TmFalse(unknown)
  | _ → raise No

let rec eval t =
  try let t' = eval1 t
      in eval t'
  with No → t
```

3.5 Summary

Booleans $\mathbb{B}$ **(untyped)**

Syntax

`t`	::=		*(terms...)*
		`true`	*constant true*
		`false`	*constant false*
		`if t then t else t`	*conditional*
`v`	::=		*(values...)*
		`true`	*value true*
		`false`	*value false*

Evaluation $(\mathtt{t} \longrightarrow \mathtt{t}')$

$$\mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_2} \qquad \text{(E-BOOLBETAT)}$$

$$\mathtt{if\ false\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_3} \qquad \text{(E-BOOLBETAF)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}} \qquad \text{(E-IF)}$$

Arithmetic expressions $\mathbb{B}$ $\mathbb{N}$ **(untyped)**

New syntactic forms

`t`	::=	...	*(terms...)*
		`0`	*constant zero*
		`succ t`	*successor*
		`pred t`	*predecessor*
		`iszero t`	*zero test*
`v`	::=	...	*(values...)*
		`0`	*zero value*
		`succ v`	*successor value*

New evaluation rules $(\mathtt{t} \longrightarrow \mathtt{t}')$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{succ\ t_1} \longrightarrow \mathtt{succ\ t_1'}} \qquad \text{(E-SUCC)}$$

$$\mathtt{pred\ 0} \longrightarrow \mathtt{0} \qquad \text{(E-BETANATPZ)}$$

$$\mathtt{pred\ (succ\ v)} \longrightarrow \mathtt{v} \qquad \text{(E-BETANATPS)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{pred\ t_1} \longrightarrow \mathtt{pred\ t_1'}} \qquad \text{(E-PRED)}$$

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as `initramfs`. This file system contains necessary drivers and kernel modules that must be installed to support the ***real*** root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the `systemd` process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt. In Section 11.5.2, we describe the boot process for Windows.

It is worthwhile to note that the booting mechanism is not independent from the boot loader. Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.

The boot process for mobile systems is slightly different from that for traditional PCs. For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders. The most common Android boot loader is LK (for "little kernel"). Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system. However, whereas Linux discards the `initramfs` once all necessary drivers have been loaded, Android maintains `initramfs` as the root file system for the device. Once the kernel has been loaded and the root file system mounted, Android starts the `init` process and creates a number of services before displaying the home screen.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into **recovery mode** or **single-user mode** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

2.10 Operating-System Debugging

We have mentioned debugging from time to time in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

2.10.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log fil** to alert system administrators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the "core" in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process at the time of failure.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel's memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

2.10.2 Performance Monitoring and Tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. Tools may be characterized as providing either ***per-process*** or ***system-wide*** observations. To make these observations, tools may use one of two approaches—***counters*** or ***tracing***. We explore each of these in the following sections.

2.10.2.1 Counters

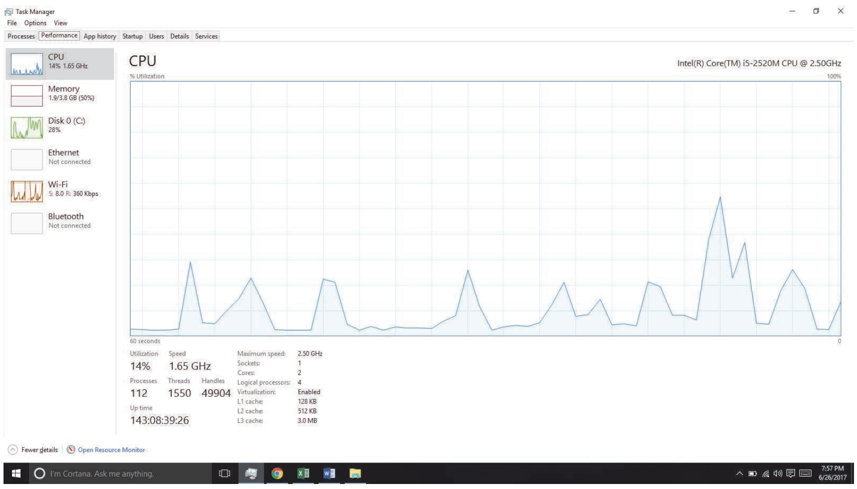
Operating systems keep track of system activity through a series of counters, such as the number of system calls made or the number of operations performed to a network device or disk. The following are examples of Linux tools that use counters:

Per-Process

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

System-Wide

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks



CPU 14% 1.65 GHz
Memory 1.9/3.8 GB (50%)
Disk 0 (C:) 28%
Ethernet Not connected
Wi-Fi S: 8.0 R: 360 Kbps
Bluetooth Not connected

Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz

Utilization 14%
Speed 1.65 GHz
Processes 112
Threads 1550
Handles 49904
Up time 143:08:39:26

Maximum speed:	2.50 GHz
Sockets:	1
Cores:	2
Logical processors:	4
Virtualization:	Enabled
L1 cache:	128 KB
L2 cache:	512 KB
L3 cache:	3.0 MB

Figure 2.19 The Windows 10 task manager.

Most of the counter-based tools on Linux systems read statistics from the `/proc` file system. `/proc` is a “pseudo” file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The `/proc` file system is organized as a directory hierarchy, with the process (a unique integer value assigned to each process) appearing as a subdirectory below `/proc`. For example, the directory entry `/proc/2155` would contain per-process statistics for the process with an ID of 2155. There are `/proc` entries for various kernel statistics as well. In both this chapter and Chapter 3, we provide programming projects where you will create and access the `/proc` file system.

Windows systems provide the **Windows Task Manager**, a tool that includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager in Windows 10 appears in Figure 2.19.

2.10.3 Tracing

Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.

The following are examples of Linux tools that trace events:

Per-Process

- `strace`—traces system calls invoked by a process
- `gdb`—a source-level debugger

System-Wide

- `perf`—a collection of Linux performance tools
- `tcpdump`—collects network packets

Kernighan's Law

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Making operating systems easier to understand, debug, and tune as they run is an active area of research and practice. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss BCC, a toolkit for dynamic kernel tracing in Linux.

2.10.4 BCC

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument their interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This toolset must also have a minimal performance impact—ideally it should have no impact when not in use and a proportional impact during use. The BCC toolkit meets these requirements and provides a dynamic, secure, low-impact debugging environment.

BCC (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems. BCC is a front-end interface to the eBPF (extended Berkeley Packet Filter) tool. The BPF technology was developed in the early 1990s for filtering traffic across a computer network. The "extended" BPF (eBPF) added various features to BPF. eBPF programs are written in a subset of C and are compiled into eBPF instructions, which can be dynamically inserted into a running Linux system. The eBPF instructions can be used to capture specific events (such as a certain system call being invoked) or to monitor system performance (such as the time required to perform disk I/O). To ensure that eBPF instructions are well behaved, they are passed through a **verifie** before being inserted into the running Linux kernel. The verifier checks to make sure that the instructions do not affect system performance or security.

Although eBPF provides a rich set of features for tracing within the Linux kernel, it traditionally has been very difficult to develop programs using its C interface. BCC was developed to make it easier to write tools using eBPF by providing a front-end interface in Python. A BCC tool is written in Python and it embeds C code that interfaces with the eBPF instrumentation, which in turn interfaces with the kernel. The BCC tool also compiles the C program into eBPF instructions and inserts it into the kernel using either probes or tracepoints, two techniques that allow tracing events in the Linux kernel.

The specifics of writing custom BCC tools are beyond the scope of this text, but the BCC package (which is installed on the Linux virtual machine we provide) provides a number of existing tools that monitor several areas of activity in a running Linux kernel. As an example, the BCC `disksnoop` tool traces disk I/O activity. Entering the command

```
./disksnoop.py
```

generates the following example output:

```
TIME(s)           T      BYTES      LAT(ms)
1946.29186700     R      8             0.27
1946.33965000     R      8             0.26
1948.34585000     W      8192          0.96
1950.43251000     R      4096          0.56
1951.74121000     R      4096          0.35
```

This output tells us the timestamp when the I/O operation occurred, whether the I/O was a Read or Write operation, and how many bytes were involved in the I/O. The final column reflects the duration (expressed as latency or LAT) in milliseconds of the I/O.

Many of the tools provided by BCC can be used for specific applications, such as MySQL databases, as well as Java and Python programs. Probes can also be placed to monitor the activity of a specific process. For example, the command

```
./opensnoop -p 1225
```

will trace `open()` system calls performed only by the process with an identifier of 1225.

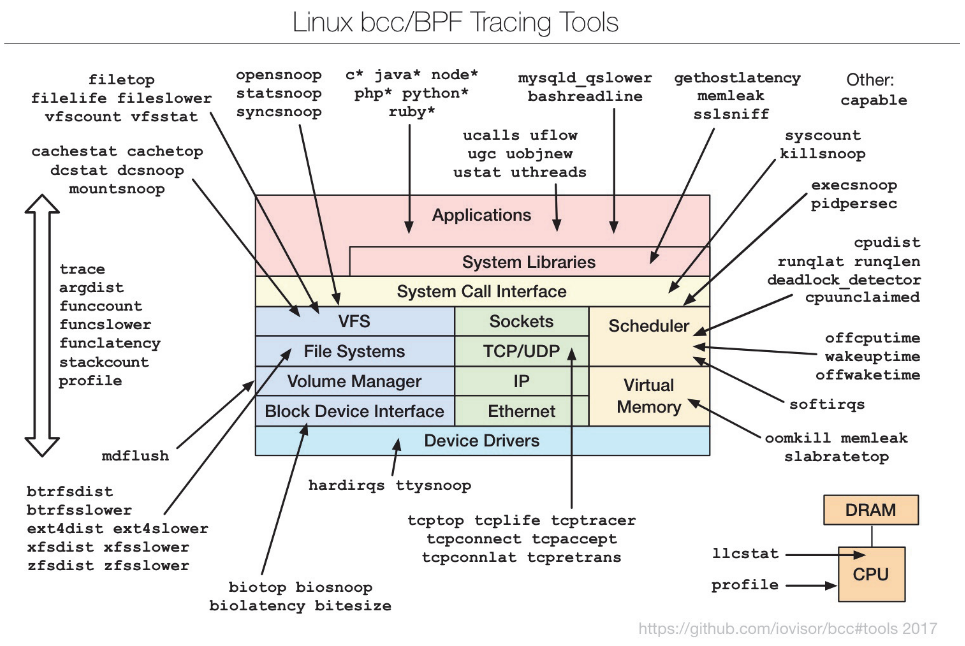


Figure 2.20 The BCC and eBPF tracing tools.

What makes BCC especially powerful is that its tools can be used on live production systems that are running critical applications without causing harm to the system. This is particularly useful for system administrators who must monitor system performance to identify possible bottlenecks or security exploits. Figure 2.20 illustrates the wide range of tools currently provided by BCC and eBPF and their ability to trace essentially any area of the Linux operating system. BCC is a rapidly changing technology with new features constantly being added.

2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some suc-