Today's Topics

Last Week
• Context free parsers and how they recognize the syntactic structure of input programs

This Week
• Semantics - the meaning of program structures
• Kinds of semantics
• Abstract machines, run time models
• Stack model of expression evaluation, scopes and variables
Semantics – The Meaning of Programs

• The “meaning” of a segment of code is called its *semantics*

• Formal *specifications* of semantics take several forms:
  
  • *Denotational* – Models the execution of a program using mathematical functions that map one state of a simple memory machine model to the next state
    
    • Each syntactic structure is mapped ("denotes") a function expression to be evaluated

    • *Compositional*, in that the function representing a given syntactic construct is based on the functions representing each of the components

    • Good for *specifying*, comparing and *reasoning* about programming languages and program equivalence
Semantics – The Meaning of Programs

- Formal specifications of semantics take several forms:
  - **Axiomatic** – Program structures are mapped to a set of logical predicate transformers, used to formally prove the correctness of the program with respect to a pre-/post-condition specification
    - Good for proving programs correct
  - **Operational** – Based on the idea of an abstract machine, designed to reflect a real machine-like low level semantic basis to which language statements are mapped
    - Good for implementing compilers and interpreters
Operational Semantics

- *Operational* or *interpretive* semantics are best suited to a compiler, since they are based on a low-level machine model much like a real machine.

- An *abstract machine* is an ideal computer where the machine level instructions are tailored to reflect the *low level semantics* of the general class of languages being described.

- Understanding of the meaning of the *statements* of the language is based on understanding what the equivalent sequences of *abstract machine instructions* do.

- Abstract machine instructions form a kind of *bridge*, because they are *simpler* and *lower level* than the language being compiled, but not as detailed and low level as a *real machine*.

- The abstract machine represents a *half-way point* between the language being compiled and the target real machine - the abstract machine avoids *machine dependent* issues such as number of registers, memory addressing modes, etc.
The Abstract Machine as Bridge

• To translate programs for execution on a target machine, the compiler must:
  1. **Analyze** the input program’s meaning in terms of the **abstract machine** (a.k.a. "virtual machine")
  2. **Implement** the abstract machine meaning of the program in terms of the **target machine**

• The abstract machine may actually be **literally** used in the compiler (e.g., **PT T-code**, **GNU abstract register machine**)

• Or it may just be used as a **model** for the compiler writer while the code generator of the compiler is being developed
Abstract Machine - Expressions

• Since the first FORTRAN compilers, expression execution has been modeled using an expression stack or evaluation stack (ES)

• Operands are pushed onto the stack from memory (models variable reference)

• The top value may be popped from the stack to memory (models assignment)

• Operations act on the top elements of the stack - the appropriate number of elements are popped from the top of the stack and the result is pushed onto the stack

• Some languages use this model directly in the language - most notably Forth, Postscript and Acrobat PDF
Expression Stack (ES) Evaluation

\[ x := 1 + 2 \]

push 1
push 2
add
pop x

\[ x := (5 + 7) \times 9 \]

push 5
push 7
add
push 9
multiply
pop x
Expression Stacks and Postfix

- The machine language interpreted by expression stacks is exactly *postfix* (a happy coincidence? I think not ...)

- As a postfix stream is read, *operands* are pushed, *operators* are evaluated

- Thus the postfix notation leaving the parser serves two purposes:
  1. It encodes the *parse tree*, resolving precedence and operation order for following phases of the compiler
  2. It is directly convertible to code for the *abstract machine*

\[
(x+y) \times z \rightarrow x \ y \ + \ z \ * \\
a+b\times c \rightarrow a \ b \ c \ * \ +
\]

push x push a
push y push b
add push c
push z multiply
multiply add
Expressions in T-Code

• The PT abstract machine is based on an expression stack and is called T-code

• Push operations of the PT T-code machine are called Literal operations - e.g., \texttt{tLiteralAddress}, \texttt{tLiteralInteger}

• Pushing the value of a variable from memory takes two operations - the first pushes the memory address of the variable on the stack, and the second evaluates the top of the stack as an address and fetches the value from the memory location

\[ x + y \times 5 \rightarrow x \ y \ 5 \ \ast \ + \]

\begin{align*}
push \ x & \quad \text{\texttt{tLiteralAddress}} \ x \\
push \ y & \quad \text{\texttt{tLiteralAddress}} \ y \\
push \ 5 & \quad \text{\texttt{tLiteralInteger}} \ 5 \\
multiply & \quad \text{\texttt{tMultiply}} \\
add & \quad \text{\texttt{tAdd}}
\end{align*}
Expressions in T-Code

- The PT abstract machine uses the paradigm of push address (tLiteralAddress) followed by evaluate (tFetchInteger) to push the value of a variable on the ES.

- This separation of push the variable's address and then fetch its value from that address is very common in abstract machines, because both these instructions are necessary to implement procedure parameters anyway.

- By not having an additional redundant push value instruction, we reduce the number of different instructions and simplify the abstract machine.

  \[
  \text{tLiteralAddress } x \quad \text{push } \leftarrow \text{address}(x) \\
  \text{tFetchInteger} \quad \text{push } \leftarrow \text{Memory}[\text{pop}]
  \]

  replace it with the value at that address in memory.
Runtime Model – Scopes

• In most procedural languages, there are 3 kinds of variables:

  • **Static**
    • Declared *globally* (in Pascal, Turing) and have a lifetime that includes the entire time the program runs
    • All global variables in C and static local variables in C (note that the *static* keyword has two meanings in C)

  • **Automatic**
    • Variables declared locally within a block. The lifetime of the variable starts when the program enters the block and end when the program exits the end of the block. The block is called the *scope* of the variable.

  • **Dynamic**
    • Dynamically allocated and released from the heap. *new* and *dispose* in Pascal, *malloc* and *free* in C, *new* and *<nothing>* in Java. Referenced by pointers. Lifetime is from *new* to *dispose* or the end of the program.
Runtime Model – The Run Stack (RS)

- **Static** variables can be modeled as **automatic** variables whose scope is the **entire program** (so we don't consider them separately).

- **Automatic** variables (including modeled static variables) are modeled using the **Run Stack (RS)** - this is an entirely different stack from the Expression Stack (ES).

- **Dynamic** variables, such as objects in Java, require the use of a **heap**, and will not be covered in this course.

- Each time a new **scope** is entered, storage for the automatic variables is allocated (pushed) on the **Run Stack** - on exit from the scope, their storage is popped from the **RS**.

- Scopes may be entered as **nested blocks**, or by calls to a **procedure** (method).

- We consider the issues of representing scopes on the **RS** using a constructive approach, starting with a very **simple** model, and refining it incrementally.

- Remember it is a **model**, not an implementation.
The Run Stack – Pascal Example

```pascal
var x: integer;

procedure p;
  var y: integer;
  z: real;
end p;

p;

p;
```

RS
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```

```
RS
  a
  c
d
e
```
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```
The Run Stack – C Example

```c
int a;

void main()
{
    int c;
    char d;
    {
        int e;
    }
    y();
    {
        int f;
    }
}

void y()
{
    int g;
}
```
Summary

**Run Time Models**
- Semantics - the **meaning** of program structures
- Kinds of semantics - **denotational, axiomatic, operational**
- Abstract machines, run time **models**
- Expression stack (**ES**) model of **expression evaluation**
- Run stack (**RS**) model of **scopes** and **automatic variables**

**Next Time**
- Refining the **RS** model - storage **reuse**, variable **addressing**
- Then: modelling procedure **call/return**, **parameters** and **functions**