Interpreters

Interpret :: (Program, Data) --> Output

or (currying the inputs) Interpret :: Program --> Data --> Output

or (grouping the last two outputs) Interpret :: Program --> (Data --> Output)

or Interpret :: Program --> Executable (Futamura's first projection, discussed below)

"online"

Compilers

Compile :: Program --> Executable

Executable is in the language of the implementation of the Interpreter (e.g., assembly language for Java interpreter)

Executable :: Data --> Output

"offline"

Partial evaluation

A computer program can be seen as a mapping of input data to output data:

```
prog :: Istatic x Idynamic --> Output
```

The partial evaluator (also called *specializer*) transforms (prog, Istatic) into prog* :: Idynamic -> Output

prog* is called the residual program and should run more efficiently than the original program

Futamura Projections

Interesting case when prog is an interpreter for a programming language. Described in 1970s by Yoshihiko Futamura

If Istatic is source code designed to run inside an interpreter, then partial evaluation of the interpreter with respect to this data/program produces prog*, a version of the interpreter that only runs that source code.

This specialized version of the interpreter is written in the implementation language of the interpreter., i.e., the language in which the interpreter has been written (e.g., assembly language for the Java interpreter).

The source code does not need to be re-supplied to prog*

prog* is effectively a compiled version of Istatic

This is the first Futamura projection. There are three Futamura projections:

A specializer is a partial evaluator

1. **Futamura Projection 1**: Specializing an interpreter for given source code, yields an executable
    - **Interpret :: prog --> Executable**
    - *Specializing (partially evaluating) the Interpreter for the program*

Specialize1 :: (Interpret, prog) --> Executable

- *Currying the arguments*

  Specialize1 :: Interpret --> prog --> Executable

- *Grouping the last two outputs*

  Specialize1 :: Interpret --> (prog --> Executable)

2. **Futamura Projection 2**: Specializing the specializer for the interpreter (as applied in #1) yields a compiler

   - **Specialize1 :: Interpreter --> Compiler**

   - *Specializing the specializer for the Interpreter*

     Specialize2 :: (Specialize1, Interpreter) --> Compiler

   - *Currying the arguments*

     Specialize2 :: Specialize1 --> Interpreter --> Compiler

   - *Grouping the last two outputs*

     Specialize2 :: Specialize1 --> (Interpreter --> Compiler)

3. **Futamura Projection 3**: Specializing the specializer for itself (as applied in #2), yields a tool that can convert any interpreter to an equivalent compiler.

   - **Specialize2 :: Specialize1 --> InterpreterToCompilerConverter**

Take-aways from Futamura's projections

- An executable is a partial-evaluation of an interpreter for a program's source code
- The process of partial-evaluation of an interpreter for a program's source code is called *compilation*
- The execution of the partially-evaluated interpreter (for the program) should ideally be faster than the execution of the original interpreter on the program.
  - In other words, the execution of the executable should be faster than the execution of the interpreter on the program.
- The quality of the compilation is determined by the speedup obtained through this *offline* partial evaluation
  - A trivial compilation is to just store the interpreter and the program's source code as a tuple; running it would involve full evaluation of the interpreter over all arguments. But this will result in slow runtime.
  - A smarter compilation will involve substituting ("constant propagating") the input program (Istatic) into the interpreter's implementation. If the interpreter is written as an evaluation loop over each statement in the source program, then compilation will also involve unrolling the loop for the number of statements in the source program:

```
Interpret(Prog, Data):
    for Statement in Prog:
        Data = Evaluate(Statement, Data)
    return Data

Executable(Data):
    Data = EvaluateStatement1(Data)
    Data = EvaluateStatement2(Data)
    Data = EvaluateStatement3(Data)
    ...
```

```
Data = EvaluateStatementN(Data)
return Data
```

Can we do better?

- Local Optimizations: Specialize the code for EvaluateStatementI. Recall that Data is not available at compile time (Idynamic), but StatementI is available (Istatic)
- Caching the generated code (An optimization that is very effective for code with loops):
  - The evaluation of a statement involves (a) generation of machine code for the Statement and (b) execution of the generated machine code on the Data
  - If one statement, say StatementI is executed several times (typical programs execute the same statement millions to billions of times), cache the generated machine code, so we do not need to regenerate the same machine code repeatedly.
    - Static compilation, also called *Ahead-of-time* Compilation: Cache the generated machine code for all statements, irrespective of whether they execute once or multiple times.
      - *Pro*: Offline, zero compilation cost at runtime.
      - *Con*: Compile + Execute may be slower (or at best equal cost) than plain interpretation for some programs, e.g., programs which have no loops.
      - *More serious con*: Optimized compilation is expensive. For a given time budget for compilation, we do not have enough information to decide how much time to spend on optimizing which part of the program. e.g., the most executed part of the program should be optimized the most, while dead-code needs no optimization. This information is not available during ahead-of-time compilation, and can only be estimated using approximation heuristics.
    - Dynamic compilation, also called *Just-in-time* Compilation:
      - Generate machine code at runtime.
      - Maintain a cache of generated code for each program fragment.
      - Cache can be limited size. Need cache-replacement policy (e.g., Least Recently Used LRU).
      - Do not need to re-generate code on cache hit.
      - *Pro:* Can spend more effort on the program fragments that are executed most (*hot regions*).
      - *Pro:* If the generated code is much bigger than the source program, then storage requirements are smaller in JIT compilation. Real problem for Android's JVM. e.g., JIT compiled Facebook App on Android is 90MB in size, while AOT compiled Facebook App takes 250MB. For a long time, Samsung hard-codeed Facebook to be JIT compiled, while some other apps were AOT compiled.
      - *Serious con*: Pay compilation cost at runtime.
        - However, this is not a very serious con for typical applications that are relatively small but run for a long time (due to loops); typical JIT compilation times for most applications are significantly smaller than typical runtimes.
        - Often the advantages of JIT optimization outweigh the JIT costs.
        - With multi-core processors, we can use the additional processors to do the JIT compilation/optimization in parallel.
- Global optimizations: Optimizations that span multiple statements (or program fragments) are often possible.
  - Because the programmer wrote sub-optimal code. It is our job to optimize it now. e.g., the programmer wrote x=1; x=2;
  - Because the machine representation may be richer than the source code syntax. e.g., the C syntax has multiply and add as separate operations, but the machine may have a single instruction to do multiply-and-add in one go.
  - Because some optimizations enable other optimizations. e.g., constant substitution can trigger more constant substitution.
    - *Caveat*: On the other hand, it is also possible for some optimizations to *preclude* other optimizations. e.g., replacing multiplication with shift may preclude the use of

multiply-and-add instruction.
- And several other reasons...
  - Generating the optimal implementation for a given program specification is an undecidable problem in Turing's model of computation, and an NP-Hard problem in the finite model of computation.

Why study compilers? Can't I just take them for granted and forget about them?

If you ask me, compilers are the most exciting research area in computer systems (and perhaps in the whole of computer science) currently, and are likely to remain so for several years to come.

Two technology trends that show the rising importance of compilers in Computer Science:

1. *Moore's law is running out of steam*
   - Gordon Moore, the co-founder of Fairchild Semiconductor and Intel, in his 1965 paper, described a doubling every year in the number of components per integrated circuit for at least the next decade.
   - In 1975 (after one decade), he revised it to doubling every two years.
   - Usually meant faster and more powerful computers, but speeds saturated in early 2000s, necessisating the need towards multi-core computing.
   - In 2015, Intel CEO, Brian Krzanich, revised the estimate to "doubling every two and a half years".
   - Physical limits to transistor scaling, such as source-to-drain leakage, limited gate metals, and limited options for channel material have been reached. While scientists are still exploring more ways to scale the transistors further (e.g., electron spintronics, tunnel junctions, etc.), it is safe to assume that all good things come to an end.
   - Moore's law meant that architects and computer system engineers did not have to work too hard
     - The process and materials engineers are already making great improvements. We are too busy catching up with that.
       - *Consequence*: Our OS and programming language stacks are quite similar to what they were in 1960s-70s. Seems absurd to imagine that this will continue as-is for the next 50 years.
     - Need innovations at the architecture level: perhaps we need innovative computing units not limited to sequential execution of simple instructions, or perhaps something else
       - If you read research papers in the architecture community, it is evident that several innovative ideas to improve performance have been proposed over the past 20 years or so.
       - Yet, these ideas have not become mainstream.
         - The primary stumbling block is: we do not know how to automatically and efficiently map program specifications written by humans to these new innovative architectural constructs.
         - Humans find it hard to directly code using these architectural constructs/abstractions, as they are not intuitive
         - Today's compilers are too fragile
           - The original designs for compilers were meant to handle simple opcodes (e.g., add, multiply, etc.), which were easily generated using similar constructs used by humans in their programs.
           - Over the years, this design the compiler has been loaded with several more requirements: vectorization, more programming constructs, more opcodes, parallel constructs and implementation, and so on...
             - Result: Over 14 million lines of code in GCC. Has increased at the rate of two million SLOC every three years. I predict that it will continue to grow at this rate (or higher) for the next decade, if the compiler design is not changed. You can call this the Compiler Complexity Law (or Sorav's Law if you will ;-).
2. *Program complexity is increasing*
   - We expected programs to just perform arithmetic computations in 1960s. Today we expect them to talk to us, write English books, and perhaps even develop a compiler!

    - This has given rise to programming in higher levels of abstraction
        - Higher levels of abstraction was about constructs like objects, automatic memory management constructs, etc. (e.g., Java, C++), in late 1980s and 1990s.
        - In 2000s, this was about abstractions for distributed data-intensive applications, e.g., Map, Reduce, Sort, Filter, etc.
        - In 2010s, this was about machine-learning constructs like Neuron and Connections, e.g., TensorFlow.
        - Several other domain-specific abstractions/languages have emerged --- e.g., image and video processing, web programming, UI programming, networking, etc.
    - Higher levels of abstraction increase the gap between the programming language and the architectural abstractions, making the search space for optimal implementations even larger. Thus compilers assume a central role in this setting.

Some more crazy ideas that seem to be going around these days

- Natural language processing can be modeled as a compilation problem? I do not believe this, but I also do not believe several other things that machine-learning people say ;-)
- Almost every problem in computer science is a compilation problem. For example, can't I just say that I need a web-server (in some very high-level syntax), and can't a compiler generate a highly optimized implementation of a webserver (using multiple machines in a data-center) for me?

Some history

- 1954 : IBM develops the 704 (the first commercially-successful machine)
    - Found that software costs exceeded the hardware costs by a lot
- 1953 : John backus introduced something called "Speedcoding"
    - A small language that was much easier to program than directly programming the machine
    - 10-20x slower (interpreted)
    - Interpreter itself took 300 bytes of memory (which was 30\% of all available memory).
    - Did not become popular
- Most common use of computers was for evaluating arithmetic formulae. John Backus, took his Speedcoding experience, to develop a higher-level program language for Formulas, and automatically "Translate" these "Formulas", in a language called "ForTran". The compiler took three years to develop (during 1954-57).
- By 1958, 50\% of all programs were written in ForTran1 --- huge success!
- The first Fortran compiler led to a huge body of theoretical work. Modern compilers preserve the outline of Fortran1!
    - Compilers is a field that involves a mixture of theory and systems in quite an intense way.

Outline of the Fortran1 compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
    - Types, scope, etc.
4. Optimization
5. Code generation (translation)

Steps in a compiler in some more detail

Explain using an analogy to how humans understand written english.

- Understanding English: First step is to recognize words (lexical analysis)
    - Words are the smallest unit above letters
    - Example: `This is a sentence.`
        - Background computation: identify separators (white spaces), capitals, punctuation marks, etc., to identify words and sentence.

        - Counterexample: `ist hisa sen tence.` is much harder to read. Hence, first step is to group letters into words (or tokens).
    - `if x == y then z = 1; else z = 2;`
        - Need to group these characters into tokens.
        - The language keywords (if, then, else) are each represented as a single token.
        - Each identifier (x, y, z) is also identified as a single token.
        - Each operator (=, ==) is identified as a single token. Now, how do we know that "==" represents a single token or is a combination of two "=" tokens? We will see this when we discuss the lexical analyzer.
        - Delimiters like white-space and semi-colon, are also recognized as separate tokens. Multiple white-space is recognized as a single token in this language, for example.
- English language: the next step is to understand sentence structure (parsing)
    - Parsing = diagramming sentences as trees
    - `This line is a longer sentence`
        - `This`: Article
        - `line`: Noun
        - `is`: Verb
        - `a`: Article
        - `longer`: Adjective
        - `sentence`: Noun
        - Next level:
            - `This line` : subject
            - `is` : verb
            - `a longer sentence` : object
        - Next level:
            - `This line is a longer sentence` : subject verb object : sentence
    - `if x == y then z = 1; else z = 2;`
        - `x==y` : relation
        - `z = 1;` : assignment
        - `z = 2;` : assignment
    - Next level
        - `x == y`: predicate
        - `z = 1`: then-statement
        - `z = 2`: else-statement
    - Next level
        - predicate then-statement else-statement : if-then-else
    - Is it a co-incidence that English parsing is exactly similar to program parsing?
- English language: the next step is to understand the "meaning" of the sentence (semantic analysis)
    - This is too hard!
    - Reduces to the equivalence checking problem in computers, which is undecidable!
    - Compilers perform limited semantic analysis to catch inconsistencies, but they do not know what the program is supposed to do.
    - Variable binding : an example of semantic analysis
        - `Jack said Jerry left his assignment at home`
            - Whom does `his` refer to? Jack or Jerry?
        - Even worse: `Jack said Jack left his assignment at home`
            - How many people are involved? Could be 1, 2, or 3.
        - These are examples of *ambiguity* in the english language syntax.
        - Such ambiguities are resolved in programming languages through strict variable binding rules. `{ int Jack = 3; { int Jack = 4; cout << Jack; } }`
            - Common rule: bind to inner-most definition (inner definition hides outer definition)
    - Type analysis: another example of semantic analysis
        - `Jack left her homework at home`
            - Assuming Jack is male, this is a type mismatch between `her` and `Jack`; we know they are different people.
- English language: optimization is perhaps a bit like editing (not a strong correlation)
    - Automatically modify programs so that they
        - Run faster

        - Use less memory
        - Reduce power
        - Reduce network messages or database queries
        - Reduce disk accesses, etc.
    - Optimization requires precise semantic modeling; several subtleties emerge in this context.
        - Can `for (unsigned i = 0; i < n + 1; i++) {}` be changed to `for (unsigned i = 0; i <= n; i++) {}`? Answer: No!
        - Can `for (int i = 0; i < n + 1; i++) {}` be changed to `for (int i = 0; i <= n; i++) {}`? Answer: Yes!
        - Can `(2*i)/2` be changed to `i`? Answer: No!
        - Can `Y*0` be changed to `0`? Answer: No for floating point!
- English language: Translation into another language (code generation)
    - Usually generate code for assembly language in compilers.
- The overall structure of almost every compiler adheres to this outline, yet proportions have changed.
    - Sorted in order of size/complexity in Fortran: Lexer, Parser, CG, Optimizations, Semantic analysis
    - Sorted in order of size/complexity in today's compilers: Optimizations (by a huge margin), Semantic analysis (deeper than before), CG (rather small), Parser, Lexer

Economy of Programming Languages

- Why are there so many (hundreds to thousands) programming languages?
    - Application domains have distinctive/conflicting needs
        - Scientific computing
            - Need rich support for floating-point operations, arrays, and parallelism. e.g., Fortran
        - Business applications
            - Need support for persistence, report generation facilities, data analysis. e.g., SQL
        - Systems programming
            - Need support for fine-grained control over resources, real-time constraints. e.g., C/C++

      Different languages make it easier for human programmers to *efficiently* encode different types of logic, and require different types of optimization support! Somewhat related to the fragility of our current compilers.

Lexical analysis

```
if (x == y)
  z = 1;
else
  z = 2;
```

will look like this to the lexical analyzer:

```
\tif (x == y)\n\t  z = 1;\n\telse\n\t  z = 2;\n
```

The lexical analyzer has to put dividers between different units. e.g., a divider between `\t` and `if` and `<whitespace>` and `(` and so on ...

Token Class (or Class)

- In English: Noun, Verb, Adjective, ...
- In programming language: Identifier, Keyword, (,), ...

Token classes correspond to sets of strings

- Identifier: a string of letters or digits, starting with a letter.
- Integer: a non-empty string of digits
- Keyword: 'if', 'then', 'else', ';', etc.
- Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical analyzer classifies strings according to role (token class). Communicates tokens to parser.

```
lexer :: string --> [ token ]
token = <class, string>
```

Example input:

```
foo = 42
```

Example output:

```
<Id, "foo"> <Op, "="> <Int, "42">
```

In our original example

```
\tif (x == y)\n\t  z = 1;\n\telse\n\t  z = 2;\n
```

the relevant token classes are:

- Operator
- Whitespace
- Keywords
- Identifiers
- Numbers
- '('
- ')'
- ';'
- '='

Work through the example to show the resulting list of tokens

A lexical analyzer needs to

1. recognize substrings corresponding to tokens (called *lexemes*)
2. recognize token class of each lexeme to generate <token-class, lexeme>

Lexical analysis examples: Fortran

In Fortran, whitespace is insignificant:

```
VAR1
```

is exactly the same as

```
VA R1
```

Fortran idea: removing all the whitespace should not change the meaning of the program.

Loop example

```
DO 5 I = 1,25
```

Here `DO` is a keyword representing a loop (similar to `FOR` in C), `I = 1,25` represents that the iteration variable `I` ranges from `1..25`. The number `5` represents that the loop extends from the `DO` statement till the statement with label 5, including all statements that are in between.

Another example:

```
DO 5 I = 1.25
```

The only difference in this code from the previous code is the replacement of `,` with `.`. This simply means that the variable `DO5I = 1.25`, i.e., a variable has been assigned an integer (there is no loop). The problem is that just by looking at the first three characters, I cannot tell whether `DO` is a keyword or a prefix of a variable name. Hence we need a *lookahead*. In this example, a large lookahead is required. Ideally, the language design should ensure that the lookahead is small.

1. Goal: partition the string. Implemented by reading left-to-right, recognizing one token at a time
2. *Lookahead* may be required to decide where one token ends and the next token begins. We would like to minimize lookahead.

Lookahead is always required

For example, when we read the first character of the keyword `else`, we need to lookahead to decide whether it is an identifier or a keyword. Similarly, we need lookahead to disambiguate between == and =.

PL/1: keywords are not reserved.

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

This makes lexical analysis a bit more difficult -- need to decide what is a variable name and what is a keyword, and so need to look at what's going on in the rest of the expression.

In fact, there are examples where PL/1 may require unbounded lookahead!

Some of these problems exist in languages today: e.g., C++

C++ template syntax

```
Foo<Bar>
```

C++ stream syntax

```
cin  >> var
```

For a long time, you had to put a blank between two ">" signs in the template syntax. Most C++ compilers have fixed this now.

Regular Languages

Typically, language designers would like to keep lexical analysis as simple as possible. Perhaps the most simple class of languages is *Regular Languages*. Hence, lexical analysis of most programming languages can be implemented by implementing a classifier for regular languages. Regular languages are the weakest formal languages widely used, and have many applications.

Regular Expressions

```
'c' = {"c"}
```

```
\epsilon = {""}
```

Note that \epsilon (a language that has a single element, namely the empty string) is not the same as \phi (empty language).

Union: A + B = { a | a in A} U { b | b in B}

Concatenation: AB = { ab | a in A and b in B}

Iteration (Kleene closure): A* = A0 + A1 + A2 + ... A\infinity
A0 = \epsilon
Ai = A concatenated with itself i times

Definition: The regular expressions over \Sigma are the smallest set of expressions that includes the following:

```
R =    \epsilon
    |  'c'
    |  R + R
    |  RR
    |  R*
```

This is called the grammar of the regular expressions. We will learn more about grammars when we talk about parsing (not relevant at this stage).

\Sigma = {0, 1}
1* = all strings of 1s (1 + 0)1 = {11, 01}
0* + 1* = all strings of 0s UNION all strings of 1s (0 + 1)* = all strings of 0s and 1s (or all strings over the entire alphabet) = \Sigma* (special name for this language)

Many ways of writing any regular language

Regular expressions specify regular languages

Five constructs: Two base cases (\epsilon, single-character strings). Three compound expressions (union, concatenation, iteration)

Formal languages

Let \Sigma be a set of characters (an *alphabet*).

A *language* over \Sigma is a set of strings of characters drawn from \Sigma

e.g., English language: Alphabet = characters, Language = Sentences.

Another example: Alphabet = ASCII character set, Language = C programs

Meaning function *L* maps syntax to semantics. e.g., L(e) = M

Example: *e* could be a regular expression, and *M* would be the set of strings that it represents

Meaning of regular expressions:

```
L(\epsilon) = { "" }
L('c') = { "c" }
L(A + B) = L(A) U L(B)
L(AB) = {ab | a \in L(A) and b \in L(B)}
L(A*) = U(i>= 0)L(A^i)
```

Significance of the meaning function: separates syntax from semantics (we can change syntax without changing semantics; different syntax good for different languages/environments). Also, allows for redundancy: multiple syntaxes for the same semantics, e.g., some are concise while others are more efficient (optimization!)

Roman numerals vs. decimal number system: the meanings of both number systems are same, but the notation is extremely different

Example of redundancy: L(0*) = L(0 + 0*) = L(\epsilon + 00*) = ...

Thus *L* is a many-to-one function, and this is the basis of optimization. *L* is never one-to-many

Lexical Specifications

```
Keyword: "if" or "else" or "then" or ...
       : 'i''f' + 'e''l''s''e' + 't''h''e''n' + ...
       : 'if' + 'else' + 'then' + ... (shorthand)
```

Integers = non-empty string of digits

```
digit = '0' + '1' + '2' + ...
Integer = digit.digit*    (ensures that there is at least one digit, also represented as digit+)
```

Identifier = strings of letters or digits, starting with a letter

```
letter = 'a' + 'b' + ... + 'z' + 'A' + ... + 'Z' (shorthand: [a-zA-Z])
Identifier = letter(letter + digit)*
```

Whitespace = a non-empty sequence of blanks, newlines, and tabs

```
Whitespace = (' ' + '\n' + '\t')+
```

Example of a real language: PASCAL

```
digit = '0' + '1' + '2' + ...
digits = digit+
opt_fraction = ('.' digits) + \epsilon
opt_exponent = ('E'('+' + '-' + \epsilon) digits) + \epsilon
num = digits opt_fraction opt_exponent
```

Given a string *s* and a regular expression *R*, does *s \in L(R)*?

Some additional notation (for conciseness)

```
A+ = AA*
A | B = A + B
A? = A + \epsilon
[a-z] = 'a' + .. + 'z'
[^a-z] = complement of [a-z]
```

Lexical analysis

1. First step: write a regular expression for each token class (e.g., Keyword, Identifier, Number, ...)
2. Construct a giant regular expression that is formed by

   ```
   R = Keyword + Identifier + Number + ...
   or R = R1 + R2 + R3 + R4 + ....
   ```

3. For input `x1...xn`

   For 1 <= i <= n, check
   x1...xi \in L(R)

4. If success, we know that x1...xi \in L(Rj) for some j
5. Remove x1...xi from the input and go to step 3

This algorithm has some ambiguities.

1. How much input is used?

   ```
   x1...xi \in L(R)
   x1...xj \in L(R)
   i != j
   ```

   e.g., '=' vs. '==='. The answer is that we should always take the longer one. This is called "Maximal Munch". This is how we process the English language as well, for example (this is how humans work). E.g., if we see the word 'inform', we do not read it as 'in' + 'form' but as a single word 'inform'.
2. Which token is used?

   ```
   x1...xi \in L(Rj)
   x1...xi \in L(Rk)
   ```

   e.g., Keywords = 'if' + 'else' + ...
   Identifiers = letter(letter + digit)*

   Should we treat 'if' as a keyword or an identifier?

   Answer: use a priority order. e.g., Keywords higher priority than Identifiers

3. What if no rule matches?

   ```
   x1...xi \notin L(R)
   ```

   This will cause the program to terminate, as the program does not know what to do.

   Answer: do not let this happen. Create another token class called *Error* and put it last in the priority order.

   ```
   Error = all strings not in the lexical specification
   ```

Summary: a simple lexical analysis can be achieved using regular expressions + a few rules to resolve ambiguities and errors.

Good algorithms are known to do such lexical analysis (subject of future lectures)

- Require only single pass over the input
- Few operations per character (table lookup)

DFAs

- Only one possible transition for any input from any state

- No epsilon moves
- Only one path possible for any input

NFAs

- Potentially multiple possible transition for any input from any state
- Allow epsilon moves
- Accept if any path accepts

Conversion from NFA to DFA

- \epsilon-closure of an NFA state = all NFA states that are reachable through \epsilon moves from that state
- Have one state for each subset of NFA states: NFA can be exponentially smaller than the DFA!
- Execution of an NFA will always yield a subset of NFA states at the current execution point. Use the DFA state corresponding to this subset in the DFA path.
- While this conversion is exponential:
  - Typical number of states in the NFA are quite small
  - Exponential blow-up is worst-case. Typical blow-ups are much smaller.

Conversion from Regular language to NFA

- \epsilon : start state is an accepting state
- 'c' : consume one character to reach an accepting state from the start state
- A + B : add a new start state, and add \epsilon moves from the start state to the start states of A and B resp. Similarly, add epsilon moves from the accepting states of A and B to the accepting state of the new automaton
- AB : connect the accepting state of A to the start state of B through an \epsilon move.
- A* : add a start state, a loop state. Add \epsilon moves from start state to loop state, and from loop state to final accepting state. add \epsilon moves from loop state to A's start state and from A's accepting state to the loop state.
-

Implementing an automaton

- A DFA can be implemented as a 2D table T
  - One dimension : state
  - Second dimension : Input character
  - For every transition Si --a--> Sj, define T[i, a] = j

-
```
i = 0
state = 0
      while (input[i]) {
  state = T[state, input[i++]];
}
```

- Single pass, two lookups per input character (one for the input, another for the table)
- More space-efficient implementation of the table T are possible: e.g., share identical rows. Identical rows are very common, and so this results in significant compaction in the table's storage size. *Con*: Extra pointer de-reference during table lookup. Space vs. time tradeoff
- Even more space-efficient implementation: Use a table for the NFA (not the DFA): In this case, an element of the table would be a set of states. The table will be relatively small, because the number of states is limited by the number of NFA states and the size of the input alphabet. However, simulating the NFA is now more expensive, because we have to deal with sets of states (instead of single states). For each transition, we have to look at all the current set of possible states, to generate the next set of possible states. Again, this is a Space vs. Time tradeoff

In practice, lexical analysis give you configuration options to choose between DFAs and NFAs-based implementations (for the user to choose between space-time tradeoffs).

Several interesting and intuitive languages are not regular

```
{ (^i )^i | i >= 0 }
```

This is fairly representative of several programming constructs, e.g., nested arithmetic constructs, nested scopes, nested if-then-else, etc. Programming languages usually have a natural recursive structure (often this is the structure of human thinking)

Regular expressions cannot represent this language. Intuitively, this language requires maintaining an unbounded amount of memory (specified by *i*), while FAs only have a bounded amount of memory (the number of states)

Parser

Input: A sequence of tokens as output from the lexer
Output: parse tree of the program

Example input: `if x == y then z = 1; else z = 2;`

Example output level 1: `if cond then statement else statement`

Example output level 2: `if-then-else statement`

Example output level 3: `statement`

The parse tree may be implicit, may not be output explicitly by the compiler.

For parsing, we need:

1. A language for describing valid strings of tokens
2. A method for distinguishing valid from invalid strings of tokens

Recursive structure of programming languages are a good fit for Context-Free grammars

```
EXPR =   if EXPR then EXPR else EXPR;
       | while EXPR ( EXPR )
```

A CFG consists of:

- A set of terminals T
- A set of non-terminals N
- A start symbol S \in N
- A set of productions:

  ```
  X -> Y1Y2..Yn
  ```

  - X (on the left-hand side) must be a non-terminal. That's what an LHS means.
  - Yi could be a terminal, a non-terminal, or the special symbol \epsilon

Strings of balanced parantheses:

```
S -> (S)
S -> \epsilon
```

Here,

```
N = {S}
T = {'(', ')'}
```

Productions can be read as replacement rules: the RHS can replace the LHS

1. Begin with a string with only the non-terminal S
2. Replace any non-terminal X in the string by the right-hand side of some production X -> Y1..Yn
3. Repeat (2) until there are no non-terminals

In other words, at any step of this *derivation* we perform the following replacement:

```
X1X2X3..Xi X Xi+1..Xn --> X1X2..Xi Y1Y2..Yk Xi+1..Xn
```

A derivation is of the form:

```
S --> ... -> \alpha0 --> ... --> \alphaN
N >= 0
```

Let G be a CFG with start symbol S. L(G) of G is:

```
{a1...an | ai \in T and S ---*--> a1...an}
```

Terminals are called so because there are no rules for replacing them

For a parser, the terminals are the tokens

A fragment of C (this notation is a more concise form of writing the productions, and is used in actual tools)

```
selection_statement:   IF '(' expression ')' statement ELSE statement
                     | IF '(' expression ')' statement
                     | SWITCH '(' expression ')' statement;

iteration_statement:   WHILE '(' expression ')' statement
                     | DO statement WHILE '(' expression ')' ';'
```

```
                    | FOR '(' expression_statement expression_statement ')' statement
                    | FOR '(' expression_statement expression_statement expression ')' statement
                    | FOR '(' declaration expression_statement ')' statement
                    | FOR '(' declaration expression_statement expression ')' statement;


expression_statement:   ';'
                     | expression ';';

statement:   labeled_statement
           | compound_statement
           | expression_statement
           | selection_statement
           | iteration_statement
           | jump_statement;
```

Some elements of this language?

Another example: arithmetic expressions

```
E --> E + E
E --> E * E
E --> ( E )
E --> Id
```

The idea of a CFG is a big step.

- Membership in a language is a yes/no answer. Also need a parse-tree
- Must handle errors gracefully. Give feedback to the programmer
- Need an implementation of the CFGs (e.g., bison generates an implementation given a grammar)

Form of the grammar is important

- Many grammars generate the same language
- Parsing tools are sensitive to the grammar. Only some grammars are parsable by some tools. Will see examples later

A derivation can be drawn as a tree

- Start symbol is tree's root
- For a production X --> Y1..Yn, add children Y1..Yn to node X

Example: `id * id + id`. Generate the derivation tree for the grammar of arithmetic expressions. This derivation tree is called the parse tree of this string. While building the tree, also write the derivation rules that have been used at each step to the left.

A parse tree has

- terminals at the leaves
- non-terminals at the interior nodes

An in-order traversal of the parse tree yields the original input

Multiple parse trees are possible for the same input (even for this example)

The parse tree shows the association of the operations, the original input does not.

The example we discussed is a *leftmost-derivation*: at each step, replace the left-most non-terminal. There is an equivalent notion of a *rightmost-derivation*.

A derivation defines a parse tree. But one parse tree can have multiple derivations. The leftmost and rightmost derivations are important from the point-of-view of parser implementation (as we will see later).

Note that the leftmost and rightmost derivations have the same parse tree. This is just about the order in which the branches of the parse tree were added.

Ambiguity in CFGs

Example: id*id + id

This string has two parse-trees in the grammar of arithmetic operations: (id * id) + id AND id * (id + id)

A grammar is *ambiguous* if it has more than one parse tree for some string.

- Equivalently, there is more than one right-most or left-most derivation for some string

Ambiguity may leave the meaning of the program ill-defined.

Several ways to handle ambiguity: One method is to rewrite the grammar unambiguously (by "stratifying" it)

```
E --> E' + E | E'
E' --> id * E' | id | (E) * E' | (E)
```

Show the unique left-most derivation for `id*id + id` using this new grammar.
Enforces precedence of * over +. E can only generate sum expressions over E'.

Another example:

```
E --> if E then E | if E then E else E | ...
```

String : if E1 then if E2 then E3 else E4
Ambiguity: which "if" does the "else" match with?
Typical rule: "else" matches the closest unmatched "if" (i.e., which does not already have an associated "else")

```
E -->   MIF   /* all if are matched with else */
         | UIF   /* some if is unmatched */

MIF -->   if E then MIF else MIF
           | ...

UIF -->   if E then MIF else UIF  /* the then expression is MIF, because if it were UIF then this else should have matched it! */
           | ...
```

Show the two parse trees, and show which one is rejected by this new grammar.

Impossible to convert automatically an ambiguous grammar to an unambiguous one

Used with care, ambiguity can sometimes simplify the grammar

- Sometimes allows more natural definitions
- We need disambiguation mechanisms

Most languages and parsing tools

- Use the more natural (ambiguous) grammar
- Along with disambiguating declarations

Example: E -> E + E | E * E | Int
Precedence: %left + (means that the "+" operator is left associative)
Precedence: %left * (means that the "*" operator is left associative)
The order of this specification says which one has higher precedence in case of ambiguity
Show how these precedence rules eliminate one of the parse trees

The parser will use these declarations to decide which move to take in case of ambiguity.

Error handling

Purpose of a compiler is:

- Detect non-valid programs
- Translate the valid ones

Many kinds of possible errors (e.g., in C):

- Lexing errors. e.g., abc$abc, detected by lexer
- Syntax errors. e.g., abc * % 123, detected by parser
- Semantic errors. e.g., int x; y = x(3), detected by type checker
- Correctness. e.g., quicksort. detected by tester/user. out of scope for a compiler

Error handlers should

- Report errors accurately and clearly
- Recover from an error quickly
- Not slow down compilation of valid code

Error handling modes

- Panic
- Error production
- Automatic local and global correction. Popular area of research at some point, but not mainstream.

Panic mode is the simplest

- When an error is encountered, discard tokens until a clear role is found
- Continue from there

Looking for "synchronizing tokens", typically the statement or expression terminators (e.g., ";" in C)

Example: (1 + + 2) + 3 The parser is going to get stuck at the second +. In panic mode recovery, skip ahead to next integer and then continue. (In this example, it would restart at 2, and treat it as 1 + 2, and then it would parse fine).

Bison: use the special terminator `error` to describe how much input to skip (error productions)

```
E --> int | E + E | (E) | error int | (error)
```

The fourth production says that if you see an error while trying to parse E, then the `error` symbol will match all the input until an int.

Similarly, the fourth production says that if you encounter an error while you are inside a paranthesized expression, you can discard everything until the closing bracket.

Error productions

- Specify known common mistakes in the grammar
- Example:
  - Write 5x instead of 5*x
  - Mathematicians complained that they were used to writing 5x but compiler gives parse error
  - Response: add a production E -> EE
- Disadvantage:
  - Complicates the grammar
  - Promotes mistakes (by making the syntax less restrictive)

- Common response by production compilers: warn about such usage but accept it anyway. *Warnings*

Error correction: find a "nearby" program. Notion of "edit distance"

- Try token insertions or deletions upto a certain edit distance exhaustively.
- Cons: hard to implement, significant slowdown in compilation, "nearby" is not necessarily the intended program
- Common response by production compilers today: suggestions to the programmer, e.g., OCaml
- PL/C is an example of a past error-correcting compiler
  - Motivation: compilation was quite slow. could take a day to recompile
  - Hence wanted to find as many errors as possible in one compilation cycle

Abstract syntax trees

Review

- A parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- Abstract syntax trees (AST)
  - Like parse trees but ignore some details

Consider the grammar

```
E --> int | (E) | E + E
```

And the string

```
5 + (2 + 3)
```

After lexical analysis, a list of tokens

```
int5 '+' '(' 'int2' '+' 'int3' ')'
```

During parsing we build a parse tree *Draw a parse tree*root = E, second level expands E + E, third level expands the first E into int5 and the second E into (E), and so on... The parse tree is sufficient for compilation (it captures the nesting structure) but it has too much information, such as parantheses and single-successor nodes, etc. AST is a cleaned-up and more concise version of the parse tree. Parentheses are redundant because the tree structure tells us the grouping already

AST also captures the nesting structure, but it *abstracts* from the concrete syntax. It is more compact and easier to use. An important data structure in a compiler

```
(A): PLUS, (B), (C)
(B): 5
(C): PLUS (D), (E)
(D): 2
(E): 3
```

AST is an important data structure for the rest of the discussion

Recursive Descent Parsing (Top-down)

The parse tree is constructed (a) from the top, and (b) from left to right

Tokens are seen in order of appearence in the token stream

Example token stream (stream of terminals): t2 t5 t6 t8 t9

```
1 --> t2 3 9
3 --> 4 7
4 --> t5 t6
7 --> t8 t9
```

Consider the grammar

```
E --> T | T + E
T --> int | int * T | (E)
```

Token stream is: (int5)

Start with top-level non-terminal E (hence called top-down). Try rules for E *in order*. Walk through the input with an arrow left to right. Also try rules in order. As long as we are generating non-terminals, we do not know if we are on the right path or not. However if we hit a terminal, we can see if that matches our current token or not. If it does not, we can back-track and try another option (in order). In this case we go to T and then down to int, backtrack, then int * T, then backtrack, and then (E), and here '(' matches! So we might be on the right track, and we now need to advance the input pointer, and now we have to expand the non-terminal E. Again we will start with the first production T. And we will eventually match the input string with some production, and we are done.

General algorithm for Recursive Descent Parsing

Let TOKEN be the type of the tokens: e.g., INT, OPEN, CLOSE, PLUS, TIMES, ... Let `next` point to the next token in the input string (the arrow).

Define boolean functions that check for a match of:

- A given token terminal

```
      bool term(TOKEN, tok) { return *next++ == tok;  }//next pointer is incremented regardless!
Try the nth production of non-terminal S
bool Sn() { ... } //will succeed if input matches the nth production of S
Try all productions of S
bool S() { ... } //will succeed if input matches any production of S
```

Running on our example, functions for non-terminal E

- For production `E --> T`

```
bool E1() { return T(); }
```

- For production `E --> T + E`

```
bool E2() { return T() AND term(PLUS) AND E(); }
```

- For all productions of `E` (with backtracking)

```
bool E() {
   TOKEN *save = next;
         return    (next = save, E1())
          || (next = save, E2());
}
```

  The first rule that returns true will cause a return (we will not evaluate the remaining rules due to semantics of logical OR ||)
  `next` pointer needs to be saved for backtracking

Functions for non-terminal T

- For production `T --> int`

```
bool T1() { return term(INT); }
```

- For production `E --> int * T`

```
bool T2() { return term(INT) AND term(TIMES) AND T(); }
```

- For production `E --> (E)`

```
bool T3() { return term(OPEN) AND E() AND term(CLOSE); }
```

```
bool T() {
   TOKEN *save = next;
         return    (next = save, T1())
          || (next = save, T2())
          || (next = save, T3());
}
```

The first rule that returns true will cause a return (we will not evaluate the remaining rules due to semantics of logical OR ||)
`next` pointer needs to be saved for backtracking

To start the parser:

1. Initialize `next` to first token
2. Invoke E()
   - Show full execution of E() on the example string

Can implement by hand, e.g., TinyCC

Left Recursion : the main problem with Recursive Descent Parsing

Consider

```
S --> S a
```

```
bool S1() { return S() AND term(a); }
bool S() { return S1(); }
```

This will go into an infinite loop for parsing *any* input.

The problem is that this grammar is left recursive

```
S -->+ S \alpha    for some \alpha
```

If for some sequence of productions we end up with left recursion, recursive descent parsing will just not work

Consider the left-recursive grammar

```
S --> S a | b
```

This generates a string starting with `b` with any number of `a`s following it. It produces the sentences right-to-left, and that's why it does not work with recursive descent parsing (which works left-to-right).

We can generate exactly the same language by producing strings left to right, which is also a solution to our problem.

```
S --> b S'
S' --> a S' | \epsilon
```

In general, for any left recursive grammar

```
S --> S a1 | S a2 | ... | S an | b1 | b2 | ... | bm
```

All strings starting with one of b1..bm and continue with several instances of a1..an
Rewrite as

```
S --> b1 S' | b2 S' | ... | bm S'
S' --> a1 S' | a2 S' | ... | an S' | \epsilon
```

There are more general ways of encoding left recursion in a grammar

```
S --> A a | d
A --> S b
```

is also left-recursive because

```
S -->+ S b a
```

This left recursion can also be eliminated (see Dragon book for general algorithm).

Recursive descent

- Simple and general parsing strategy
- Left-recursion must be eliminated first
  - This can be done automatically
  - In practice however, this is done manually. The reason is that we also need to specify semantic actions with the productions used. Hence, people do elimination of left-recursion on their own, and this is not difficult to do.
- Popular strategy in production compilers. e.g., gcc's parser is a hand-written recursive-descent parser.

Predictive Parsing

- Like recursive-descent but parser can "predict" which production to use
  - by looking at the next few tokens
  - without backtracking
- Predictive parsing accepts LL(k) grammars
  - Left-to-right scanning of tokens
  - Leftmost derivation
  - k tokens to lookahead. In practice, k = 1
- In LL(1), at every step there is at most one choice for a possible production
  - At each step, only one choice of production

```
wAb --> w \alpha b , next input: token t//only one choice for A at every step
```

Example

```
E --> T + E | T
T --> int | int * T | (E)
```

Here, with one lookahead, hard to predict because:

- we cannot choose the first two productions of T by one lookahead
- the same problem exists with E, because two productions start with non-terminal T. Here it is a non-terminal, but still there is an issue, because we do not know which production to use for one lookahead

Left factoring a grammar

Basic idea: Factor out common prefix into a single production

```
E --> TX
X --> + E | \epsilon
```

Left factoring delays the decision on which production to use. Here we decide which production to use *after* we have seen `T`.

Similarly,

```
T --> int Y | (E)
Y --> \epsilon | * T
```

Left-factored grammar

E --> TX X --> +E | \epsilon T --> int Y | (E) Y --> \epsilon | * T

LL(1) parsing table (assume it is somehow given, later we will discuss how to construct this):

```
     int     *     +     (     )          $
E    TX                TX
X                +E          \epsilon   \epsilon
T    int Y             (E)
Y           *T   \epsilon     \epsilon   \epsilon
```

Empty entries represent error states

Algorithm: similar to recursive descent, except

- For the left-most non-terminal S, we look at the next input token a
- Choose the production rule shown at (S, a)

Instead of using recursive functions, maintain a stack of the frontier of the parse tree. The stack contains all the entries on the right of the top of the frontier. In other words, the top of the stack is the leftmost pending terminal or non-terminal. Terminals in the stack are yet to be matched in the input. Non-terminals in the stack are yet to be expanded.

Reject on reaching error state.

Accept on end of input or empty stack.

Pseudo-code:

```
initialize stack = <S $> and next
repeat
    case stack of
      <X, rest> : if T[X, *next] = Y1...Yn
                        then stack <-- <Y1..Yn, rest>;
                        else error();
      <t, rest> : if t == *next++
                        then stack <-- <rest>;
                        else error();
until stack == < >
```

Run this algorithm on the example

Constructing LL(1) Parsing Tables

Constructing **First sets**

Consider a non-terminal A, production A --> \alpha, and a token `t`

T(A, t) = \alpha in two cases:

1. If \alpha -->* t \beta
    - If \alpha can derive t in the first position
    - We say that t \in First(\alpha)
2. If \alpha -->* \epsilon and S -->* \beta A t \delta
    - Useful if stack has A, input is t, and A cannot derive t
    - In this case, the only option is to get rid of A by deriving \epsilon
        - Can work only if t can follow A in at least one derivation
    - We say that t \in Follow(\alpha)

Definition:

```
First(X) = {t | X -->* t \alpha} U {\epsilon | X -->* \epsilon}
```

Algorithm sketch:

1. First(t) = {t}
2. \epsilon \in First(X)
    - if X --> \epsilon
    - if X --> A1A2..An AND \epsilon \in First(Ai) for all i
3. First(\alpha) \subset First(X) if X --> A1..An \alpha and \epsilon \in First(Ai) for all i

Example

```
E --> TX
X --> +E | \epsilon
T --> int Y | (E)
Y --> \epsilon | * T
```

First set for terminals

```
First(t) = {t}
First(*) = {*}
....
```

First set for non-terminals

```
First(E) \superset First(T)
First(T) = {'(', 'int')}
First(X) = {+, \epsilon}
First(Y) = {*, \epsilon}
```

Constructing **Follow sets**

Definition

```
Follow(X) = {t | S -->* \beta X t \delta}
```

Intuition

- if `X --> A B, then First(B) \subset Follow(A) and Follow(X) \subset Follow(B)`
- if `B --> \epsilon`, Follow(X) \subset Follow(A)
- If S is the start symbol, then $ \in Follow(S)

Algorithm sketch:

1. $ \in Follow(S)
2. For each A --> \alpha X \beta,
    - First(\beta) - \epsilon \subset Follow(X)
3. For each A --> \alpha X \beta
    - Follow(A) \subset Follow(X) if \epsilon \in First(\beta)

Example

```
E --> TX
X --> +E | \epsilon
T --> int Y | (E)
Y --> \epsilon | * T
```

Running the algo

```
Follow(E) = {$, )}
Follow(E) \superset Follow(X)
Follow(X) \superset Follow(E)
(in other words Follow(X) = Follow(E))
```

Hence

```
Follow(X) = {$, )}
```

Looking at T, and the first production E --> TX

```
Follow(T) \superset First(X)
or
Follow(T) = {+} (ignoring \epsilon)
Follow(T) \superset Follow(E)    because X -->* \epsilon
Follow(T) = {+, $}
```

Looking at the second rule,

```
Follow(Y) \superset Follow(T)
Follow(T) \superset Follow(Y)
```

Follow(Y) = {+, $,)}

Looking at terminals

```
Follow('(') \superset First(E)
Follow('(') = {(, int}
Follow(')') \superset Follow(T)
Follow(')') = {+,$,)}
Follow(+) \superset First(E)
Follow(+) = {(, int}
E does not go to \epsilon, so this is it.
Follow(*) \superset First(T)
Follow(*) = {(, int}
T does not go to \epsilon, so this is it.
Follow(int) \superset First(Y) - \epsilon
Follow(int) = {*}
Follow(int) \superset Follow(T)   (because \epsilon \in First(Y))
Follow(int) = {*,+,$,)}
```

LL(1) Parsing Tables

Goal: Construct a parsing table T for CFG G

- For each production A --> \alpha in G, do:
  - For each terminal t \in First(\alpha), do T[A, t] = \alpha
- If \epsilon \in First(\alpha), for each t \in Follow(A) do T[A, t] = \alpha (for token t)
- If \epsilon \in First(\alpha), for each $ \in Follow(A) do T[A, $] = \alpha (just a special case for $)

Parsing table

```
     int     *     +     (     )          $
E    TX                TX
X                +E          \epsilon   \epsilon
T    int Y             (E)
Y           *T   \epsilon     \epsilon   \epsilon
```

Fill some entries in this table

What happens if we try and build a parsing table for a grammar that is not LL(1). Example: `S --> Sa | b`

```
First(S) = {b}
Follow(S) = {$, a}
```

In this case, both productions will appear at T[S, b]. If an entry is multiply defined (more than one productions appear in a table cell), G is not LL(1). Examples: a grammar that is ambiguous, not left-factored, left-recursive will all be not LL(1). But there are more grammars that may not be LL(1)

Most programming language CFGs are not LL(1). More powerful formulations for describing practical grammars that assemble some of these ideas in more sophisticated ways to develop more parsers

Bottom-up Parsing

Bottom-up parsing is

- more general than (deterministic) top-down parsing (but just as efficient)
- builds on ideas in top-down parsing
- hence, is the preferred method in parser generation tools

Revert to the (unambiguous) natural grammar for our example

```
E --> T + E | T
T --> int * T | int | (E)
```

Consider the string `int * int + int`

Bottom-up parsing *reduces* a string to the start symbol by inverting productions

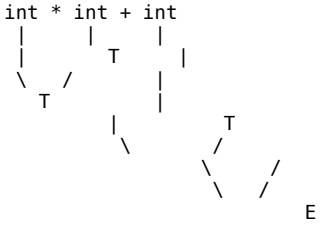
```
int * int + int               int * T + int              T --> int
int * T + int                 T + int                    T --> int * T
T + int                       T + T                      T --> int
T + T                         T + E                      E --> T
T + E                         E                          E --> T + E
E
```

Reading the productions bottom-to-top, these are the productions.
Reading them top-to-bottom, these are reductions

What's the point? The productions read backwards, trace a rightmost derivation, i.e., we are producing the right-most non-terminal at each step

Important fact #1 about bottom-up parsing
Bottom-up parsing traces a right-most derivation in reverse

Picture



Shift-reduce parsing : strategy used by bottom-up parsers

Important fact #1 has an interesting consequence:

- Let `abw` be a step of the bottom-up parse
- Assume the next reduction is by `X--> b`
- Then `w` is a string of terminals
  - because `aXw --> abw` is a step in a right-most derivation

Idea: split string into two substrings

- Right substring is as yet unexamined by parsing
- Left substring has terminals and non-terminals
- The dividing point is marked by |
  - `aX|w`
- Need two kind of moves: shift-move (discussed next) and reduce-move (already discussed)
- The shift-move moves | to the right by one character (signifying that the parser has seen this character)
- The reduce-move applies an inverse reduction to the right-hand of the left-hand string.
  - If `A --> xy` is a production, then

```
Cbxy|ijk => CbA|ijk
```

Example : `int * int + int`
Assume an oracle tells us whether to shift of reduce

```
|int * int  + int                  shift
int |* int + int                   shift
int * |int + int                   shift
int * int| + int                   shift
int *   T| + int                   reduce
T | + int                          reduce
T +| int                           shift
T + int|                           shift
T +   T|                           reduce
T +   E|                           reduce
E|                                 reduce
```

Left string can be implemented by a stack, because we only reduce the suffix of the left string. Top of the stack is |

`shift` pushes a terminal on the stack

`reduce` pops off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

In a given state, more than one action (shift or reduce) may lead to a valid parse.

If it is legal to shift or reduce, there is a *shift-reduce* conflict. Need some techniques to remove them

If it is legal to reduce using two productions, there is a *reduce-reduce* conflict. Are always bad and usually indicate a serious problem with the grammar.

In either case, the parser does not know what to do, and we either need to rewrite the grammar, or need to give the parser a hint on what to do in this situation.

Deciding when to shift and when to reduce

Example : `int * int + int`
Assume an oracle tells us whether to shift of reduce

```
|int * int  + int                    shift
int |* int + int                     shift

At this point, we could reduce.

T |* int + int                       reduce

But this would be a fatal mistake because there is no way we can reduce to E (there is no production that begins with T *).

Intuition: want to reduce only if the result can still be reduced to the start symbol (E)

Assume a right-most derivation:

S -->* \alpha X \omega --> \alpha \beta \omega

Then \alpha \beta is a handle of \alpha \beta \omega

Handles formalize the intuition: A handle is a reduction that allows further reductions back to the start symbol.

We only want to reduce at handles.

Note: we have said what a handle is, not how to find handles.

Important fact #2: In shift-reduce parsing, handles appear only at the top of the stack, never inside. Proof by induction


  1. True initially, stack is empty
  2. Immediately after reducing a handle


        ◦ rightmost non-terminal on top of stack
        ◦ next handle must be to right of rightmost non-terminal, because this is a right-most derivation.
        ◦ i.e., sequence of shift moves reaches the next handle
```

Handles are never to the left of the rightmost non-terminal, and are always at the top of the stack.

Hence shift-reduce parsers can only shift or reduce at the top of the stack. However, how to recognize handles, i.e., when to shift and when to reduce?

Recognizing handles

For an arbitrary grammar, no efficient algorithms known to recognize handles.

Heuristics to identify handles. On certain types of CFGs, heuristics are always correct.

Venn diagram: All CFGs \superset Unambiguous CFGs \superset LR(k) CFGs \superset LALR(k) CFGs \superset SLR(k) \superset LR(0)

LR(k) are quite general, but most practical grammers are LALR(k). SLR(k) are simplifications over LALR(k) [simple LR grammars]. There are grammars that are SLR(k) but not LALR(k) and so on.

\alpha is a *viable prefix* if there is a valid right-most derivation of the string: S' --> ... --> \alpha\omega --> ... --> string. e.g., \epsilon is a viable prefix

In other words, \alpha is a *viable prefix* if there is an \omega such that \alpha|\omega is a state of a shift-reduce parser. Here \alpha is the stack and \omega is the rest of the input. It can lookahead at \omega, but the parser does not know the whole think. The parser knows the whole stack.

What does this mean? A viable prefix does not extend past the right end of the handle. It's a viable prefix because it is a prefix of the handle. As long as a parser has viable prefixes on the stack, no parsing error has been detected.

An item is a production with a "." somewhere on the RHS in the production: e.g., the items for `T --> (E)` are: `T --> .(E)`, `T --> (.E)`, `T --> (E.)`, `T --> (E).`.

The only item for an \epsilon production, `X --> \epsilon` is `X --> .`. Items are often called "LR(0) items".

The problem in recognizing viable prefixes is that the stack has only bits and pieces of the RHS of productions

- If it had a complete RHS, we could reduce

These bits and pieces are always prefixes of RHS of some production(s).

Consider the input `(int)` for the grammar:

```
E --> T + E | T
T --> int * T | int | (E)
```

- Then `(E|)` is a state of a shift-reduce parse
- `(E` is a prefix of the RHS of `T --> (E)`
  - Will be reduced after the next shift
- Item `T --> (E.)` says that so far we have seen `(E` of this production and hope to see `)`

The stack may have many prefixes on RHS's:

```
Prefix1 Prefix2 Prefix3 ... Prefix(n-1) Prefix(n)
```

Let `Prefix(i)` be a prefix of RHS of `Xi --> \alpha_i`

- `Prefix(i)` will eventually reduce to X(i)
- The missing part of \alpha_(i-1) starts with X(i)
- i.e., there is a `X(i-1) --> Prefix(i-1)Xi\beta` production
- and so on.. (e.g., there is a `X(i-2) --> Prefix(i-2)X(i-1)\gamma` production.

Recursively, `Prefix(k+1) ... Prefix(n-1) Prefix(n)` eventually reduces to the missing part of \alpha(k)

Important fact #3 about bottom-up parsing: *For any grammar, the set of viable prefixes is a regular language*. The regular language represents the language formed by concatenating 0 or more prefixes of the productions (items).

For example, the language of viable prefixes for the example grammar:

```
S --> \epsilon | [S]
```

is

```
\epsilon | "["* | "["*S | "["*S"]"
```

The language of viable prefixes for the example grammar:

```
S --> \epsilon | [S] | S.S
```

is

```
X = \epsilon | "[" | "["* | "["*S | "["*S"]"
Y = "["*S.(X | Y)*
Z = X + Y
```

Conversely, In a string is parse-able through the bottom-up parser, then every potential state of the stack should always be a viable prefix.

Consider the string `(int * int)`:

- `int *|int)` is a state of a shift-reduce parse
- "(" is a prefix of the RHS of `T --> (E)`
- "\epsilon" is a prefix of the RHS of `E --> T`. This is an interesting case as we are considering "\epsilon" as a prefix too!
- "int *" is a prefix of the RHS of `T --> int * T`

Alternatively, we can represent this as a "stack of items"

```
T --> (.E)
E --> .T
T --> int * .T
```

says that

- We have seen "(" of `T --> (E)`
- We have seen "\epsilon" of `E --> T`
- We have seen "int *" of `T --> int * T`

Reading backwards, the LHS of every item becomes the RHS of the predecessor production.

In other words, every viable prefix can be represented as a stack of items, where the (n+1)th item is a production for a non-terminal that follows the "." in the nth item.

To recognize viable prefixes, we must

- Recognize a sequence of partial RHS's of productions, where
- Each partial RHS can eventually reduce to part of the missing suffix of its predecessor

Recognizing Viable Prefixes

1. Add a dummy production `S' --> S` to `G`
   This ensures that the start symbol (`S'`) is used only in one place which is the LHS of this production
2. We will construct an NFA that will behave as follows:

```
NFA(stack) = yes if stack is a viable prefix
           = no otherwise
```

3. The NFA will read the input (stack) bottom-to-top
4. The NFA states are the items of G
   - Including the extra production `S' --> S`
5. For item `E --> \alpha.X\beta`, add transition from (E --> \alpha.X\beta) --X--> (E --> \alphaX.\beta)
   - i.e., if we see X in the left state, then we go to the right state.
6. For item `E --> \alpha.X\beta` and production `X --> \gamma`, add (E --> \alpha.X\beta) --\epsilon--> (X --> .\gamma)
7. Every state is an accepting state (i.e., if the entire stack is consumed, the stack is a viable prefix)
8. Start state is (S' --> .S)

Valid Items

- Can construct a DFA from NFA using the standard subset-of-states construction
- The states of the DFA are "canonical collections of items" or "canonical collections of LR(0) items"
  - The dragon book gives another way of constructing the LR(0) items

Item `X --> \beta.\gamma` is *valid* for a viable prefix `\alpha\beta` if

```
S' -->* \alpha X \omega --> \alpha \beta \gamma \omega
```

by a right-most derivation.

After parsing `\alpha \beta`, the valid items are the possible tops of the stack of items

An item `I` is valid for a viable prefix `\alpha` if the DFA recognizing viable prefixes terminates on input `\alpha` in a state `s` containing `I`.

The items in `s` describe what the top of the item stack might be after reading input \alpha

An item is often valid for many prefixes. e.g., The item `T --> (.E)` is valid for prefixes

```
(
((
(((
((((
...
```

We can see this by looking at the DFA, which will keep looping into the same DFA state for each open paren. `Need to show the NFA and DFA construction for our example grammar, and the valid items for these string prefixes. Will need a laptop and a projector!`

Simple LR (SLR) parsing

- LR(0) Parsing: Assume
  - stack contains \alpha
  - next input is `t`
  - DFA on input \alpha terminates in state `s`
- Reduce by `X --> \beta` if
  - `s` contains item `X --> \beta`
- Shift if
  - `s` contains item `X --> \beta.t\omega`
  - equivalent to saying that `s` has a transition labeled `t`
- LR(0) has a reduce/reudce conflict if:
  - Any state has two reduce items: `X --> \beta.` and `Y --> \omega.`
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item: `X --> \beta.` and `Y --> \omega.t\delta`

SLR = "Simple LR": improves on LR(0) shift/reduce heuristics so fewer state have conflicts

Idea: Assume

- stack contains \alpha
- next input is `t`
- DFA on input \alpha terminates in state `s`

- Reduce by `X --> \beta` if
  - `s` contains item `X --> \beta`
  - `t \in Follow(X) [only change to LR(0)]`
- `Shift if`
  - `s contains item X --> \beta.t\omega`

`If there are conflicts under these rules, the grammar is not SLR`

- In other words, SLR grammars are those where the heuristics detect exactly the handles
- SLR(1) grammars work with the SLR algorithm and one lookahead

- Lots of grammars are not SLR
  - including all ambiguous grammars
- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts

- Consider the ambiguous grammar:
  - E --> E+E|E*E|(E)|int
- The DFA for this grammar contains a state with the following items:
  - E-->E*E. and E-->E.+E
  - shift/reduce conflict!
- Declaring "* has higher precedence than +" resolves this conflict in favor of reducing

- The term "precedence declaration" is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!
  - Tools allow you to print out the parsing automaton, and you will be able to see the conflict resolution in the automaton (recommended)

SLR Parsing algorithm

1. Let M be DFA for viable prefixes of G
2. Let |x1...xn$ be initial configuration
3. Repeat until configuration is S|$
   - Let \alpha|\omega be current configuration
   - Run M on current stack \alpha
   - If Mrejects \alpha, report parsing error
     - Stack \alpha is not a viable prefix
   - If M accepts \alpha with items I, let a be next input
   - Shift if X --> \beta.a\gamma \in I
   - Reduce if X --> \beta. \in I and a \in Follow(X)
   - Report parsing error if neither applies

If there is a conflict in the last step, grammar is not SLR(k). k is the amount of lookahead (in practice, k = 1)

SLR Improvements

- Rerunning the viable prefixes automaton on the stack at each step is wasteful
- Most of the work is repeated
- Remember the state of the automaton on each prefix of the stack
- Change stack to contain pairs <Symbol, DFA state>
- For a stack: <sym1,state1>...<symn,staten>
  staten is the final state of the DFA on sym1..symn
- Detail: the bottom of the stack is <any,start> where
  - any is any dummy symbol
  - start is the start state of the DFA
- Define goto[i,A] = j if statei -->A statej
- goto is just the transition function of the DFA
- Shift x
  - Push <a,x> on the stack
  - a is current input
  - x is a DFA state
- Reduce X --> \alpha
  - As before
- Accept
- Error

Action table: For each state si and terminal a

- If si has item X --> \alpha.a\beta and goto[i,a] = j, then action[i,a] = shift j
- If si has item X --> \alpha. and a \in Follow(X) and X != S', then action[i,a] = reduce X-->\alpha
- If si has item S'-->S., then action[i,$] = accept
- Otherwise, action[i,a]=error

SLR Improvements

- Let I = w$ be initial input
- Let j = 0
- Let DFA state 1 have item S'-->.S
- Let stack = <dummy, 1>
- repeat
  case action[top_state(stack),I[j]] of
  - shift k: push < I[j++], k >
  - reduce X-->A:
    - pop |A| pairs
    - push <X,goto[top_state(stack),X]>
  - accept: halt normally
  - error: halt and report error

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!
- However, we still need the symbols for semantic actions (e.g., code generation)

- Some common constructs are not SLR(1)
- LR(1) is more powerful
  - Build lookahead into the items (fine-grained disambiguation for each item, rather than just checking the follow set)
  - An LR(1) item is a pair: LR(0) item, lookahead
  - [T --> .int *T,$] means
    - After seeing T--> int * T, reduce if lookahead is $
  - More accurate than just using follow sets
  - Take a look at the LALR(1) automaton for your parser! (uses these items, but has a slight optimization over it)

- LALR automaton is composed of states containing LR(1) items, with the added optimization that if two states differ only in lookahead then we combine those states. If the resulting states (after this minimization), have no conflict in the grammar, then that grammar is LALR also.

SLR Examples

```
S' --> S
S --> Sa
S --> b

SLR parsers do not mind left-recursive grammars

The first state in the corresponding DFA will look like (\epsilon closure of the NFA state): (state 1)

S' --> .S
S --> .Sa
S --> .b

If we see a b in this state, we get another DFA state: (state 2)

S --> b.

Alternatively, if we see a S in this state, we get another DFA state: (state 3)

S' --> S.
S --> S.a

From this state, if we see a, we get (state 4)

S --> Sa.

The only state with a shift-reduce conflict is state3. Here, if we look at follow of S', we have only "$", and hence we
can resolve this conflict by one lookahead. Hence this is an SLR grammar

Another example grammar

S' --> S
S --> SaS
S --> b

Looking at the corresponding DFA: (state 1)

S' --> .S
S' --> .SaS
S --> .b

One possibility is that we see b in this state to get: (state 2)

S --> b.

Another possibility is that we see S in this state to get: (state 3)

S' --> S.
S' --> S.aS

If we get a in this state, we get (state 4)

S' --> Sa.S
S --> .SaS
S --> .b

(notice that we formed an \epsilon-closure of the first item to add more items

From here, if we get S, we get the following state: (state 5)

S --> SaS.
S --> S.aS

From here, if we get a again, we go back to state 4! If we get b, we go to state 2

The only states that have conflicts are: state3 (resolved by follow as follow(S') = $) and state5 (has a shift/reduce
conflict because a \in follow(S))

Thus this is not an SLR(1) grammar
```

Catch errors that are not caught by parsing (because many constructs are not context-free).

Examples (C):

1. All identifiers are declared
2. Types
3. ...

The set of legal programs is a subset of the parse-able programs.

Scope

- Match identifier declarations with uses
  - Important static analysis step

  Example 1:

```
        void foo (int n) {
          int a = 0, i;
    printf("a = %d\n", a);
                for (i = 0; i < n; i++) {
      int a = 1;
      printf("a + j = %d\n", a + j); //a's value is 1. where is j. error?
                }
    printf("a = %d\n", a);
  }
```

- The *scope* of an identifier is the porition of the program in which that identifier is accessible
- The same identifier may refer to different things in different parts of the program
  - Different scopes for same name don't overlap
- An identifier may have restricted scope
- Most languages have *static* scope
  - Scope depends only on the program text, not run-time behaviour. e.g., C
- A few languages are *dynamically* scoped
  - Lisp (earlier versions, now moved to static scoping), SNOBOL
  - Scope depends on execution behaviour of a program
- A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program
  - Example:

```
void foo(int x)
{
  int a = 4;
  bar(3);
}
void bar(int y)
{
  printf("a = %d\n", a); //refers to a in the closest enclosing binding in the execution of the program
}
```

- We will talk about dynamic scope later ...
- In C, identifier bindings are introduced by:
  - Function definitions (introduces method names; only allowed at top-level)
  - Struct definitions (introduces type names aka class names)
  - Variable declarations (introduces objects of certain types; objects represent regions of memory)
  - Structure field definitions (introduces objects)
  - Function argument declarations (introduces objects of certain types)
  - Typedefs (introduces type names)
- Not all identifiers follow the most-closely nested rule
  - Function definitions in C cannot be nested
  - Forward declarations are allowed for functions and variables (extern keyword). Thus a variable can be used before it is defined.
  - In C++, you can use member functions for the type-variables even if they have never been declared/defined (type-variables are the arguments to the template keyword)
  - In C++ class declaration, you can use a member variable in a method before it is defined.
  - Some languages allow identifiers to be overridden within the same scope, others don't. Some languages allow identifiers to be overridden in different nested scopes, others may not.

Symbol Tables

- Much of semantic analysis can be expressed as a recursive descent of an AST
  - *Before*: Process an AST node `n`
  - *Recurse*: Process the children of `n`
  - *After*: Finish processing the AST node `n`
- When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined
- A *symbol table* is a data structure that tracks the current bindings of the identifiers
- Example: the scope of a variable declaration is one subtree of the AST

```
{ //new scope
  int x = 4;  //declarations
  f(x);
```

```
}
Tree:
NewScope -->(left child) declarations x = 4
NewScope -->(right child) statements
```

On entry to the new scope, the new declarations will be added to the symbol table before recursing down right child (statements). On returning from the new scope, the new declarations will be removed and the old declarations of x (if any) will be restored in the symbol table.

- For implementing a symbol table, we can use a stack of scopes (assuming C89)
  - `enter_scope()`: start a new nested scope
  - `find_symbol(x)`: search stack starting from top, for the first scope that contains `x`. Return first `x` found or NULL if none found
  - `add_symbol(x)`: add a symbol to the current scope (top scope)
  - `check_scope(x)`: true if `x` defined in the current scope (top scope). allows to check for double definitions.
  - `exit_scope()`: exit current scope
- For implementing forward declarations, or C++ style use-before-define styles, one may have to make multiple passes over the AST. First pass: gather all member names. Second pass: Do the checking. In general, semantic analysis requires multiple passes (often more than two). Easier to break the analysis into simpler passes, as opposed to one big complicated pass.

Types

- What is a type
  - The notion varies from language to language
  - Typically
    - A set of values
    - A set of operations on those values
  - Classes are one instantiation of the modern notion of type
- Consider the assembly language fragment:

```
add $r1,$r2,$r3
```

What are the types of $r1, $r2, $r3? Can't say. For all you know, r1 may represent a bit-pattern that represents a string. The only thing we can say is that these have the base-type *bitvector*.

- Types restrict the set of legal programs further
  - e.g., addition on a string seems quite unlikely
  - Tradeoff: catch common mistakes (e.g., addition to string is quite unlikely), but may eliminate some more-optimized programs (we can still specify all programming functionality, just that the remaining set of programs may not be as optimized as the original set of programs).
  - Different programming languages take different tradeoff points, e.g., OCaml has immutable variables (i.e., values are written only at time of creation) while C has mutable variables (values can be written at any time).
  - Well-typed programs (or well-defined programs) are a subset of parseable programs that have all symbol uses matched with its corresponding definition.
    - A type checking logic can execute either at compile time or at runtime to disambiguate well-typed programs from not-well-typed programs
    - *or* this checking responsibility can be left to the programmer (undefined behaviour)
- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C.
  - It does make sense to add two integers
  - But both have the same assembly language implementation!
- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with correct types
  - Enforces intended interpretation of values, because nothing else will!
- Three kind of languages
  - Statically typed: All or almost all checking of types is done as part of compilation (C, Java)
  - Dynamically typed: Almost all checking of types is done as part of program execution (Scheme, Lisp, Python, Perl)
  - Untyped: No type checking. All strings in the language (e.g., CFG) are valid strings. e.g., machine code
- Another variation of typing: undefined behaviour. Accessing word beyond the length of an array is not type-checked in C. But if this happens at runtime, the program is not well-defined (or well-typed). The type-system is in programmer's head, but nobody will check it for you!
- Competing views on static vs. dynamic typing
  - Long-standing debate which is better
- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime checks
- Dynamic typing proponents say:
  - Static type systems are restrictive. Restricts the programs that you can write (even though they may be well-typed at runtime)
    - Array de-reference quite hard to verify statically. Restricting programs to only those programs that can be verified statically is a non-starter. Java decides to use runtime checks (dynamic type-checking).
  - Rapid prototyping difficult within a static type system
- Some strange things about typing
  - A lot of code is written in statically typed languages with an "escape" mechanism
    - Unsafe casts in C, Java
    - Philosophy: if the programmer knows what she is doing, let her do whatever she wants. e.g., addition on strings. If the programmer is making a mistake, the program will behave badly, but the language does not provide any guarantees (after all you took the matter in your hands)
  - People retrofit static typing to dynamically typed languages

        - Avoid runtime costs
        - Debugging
        - This can only be best-effort, no guarantees. Some dynamic checks may remain
- In C
    - The user declares types for identifiers
    - The compiler infers types for expressions
- *Type checking* is the process of verifying fully typed programs. Types are already available. We just check if the type rules are obeyed.
- *Type inference* is the process of filling in missing type information.
- Type-checking and type-interference are different terms, but people often use them interchangeably.

Type checking formalism

- Inference rules have the form
    - If Hypothesis is true, then Conclusion is true
- Type checking computes via reasoning
    - If E1 and E2 have certain types, then E3 has a certain type
- Rules of inference are a compact notation for "If-Then" statements
- The notation is easy to read with practice
- We will start with a simplified system and gradually add features
- Building blocks
    - Symbol ^ is "and"
    - Symbol => is "if-then"
    - x:T is "x has type T"
- If `e1` has type `Int` and `e2` has type `Int`, then `e1+e2` has type Int
- `(e1:Int ^ e2:Int) => e1+e2:Int`
- `(e1:Int ^ e2:Int) => e1+e2:Int` is a special case of `Hypothesis1 ^ Hypothesis2 ... ^ HypothesisN => Conclusion`
  This is an inference rule
- By tradition inference rules are written

```
|- Hypothesis ... |- Hypothesis
------------------------------
     |- Conclusion
```

- e.g., C types have hypotheses and conclusions of the form

```
|- e:T
```

- `|-` means "it is provable that ..."

Some rules

```
i is an integer literal (constant)
---------------------------------  [Int]
     |- i:int

|-e1:int    |-e2:int
---------------------------------  [Add]
     |-e1+e2:int
```

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions

```
1 is an int literal
-------------------
  |- 1:int

2 is an int literal
-------------------
  |- 2:int

|-1:int   |-2:int
-----------------
   |-1+2:int
```

- A type system is *sound* if
    - Whenever `|-e:T`
    - Then `e` evaluates to a value of type `T`
- We only want sound rules
    - But some sound rules are better than others!

      ```
      1 is an integer literal
      -----------------------
         |-1:void
      ```

      This is correct (sound) but not the most precise typing rule. e.g., it will not allow 1 to be added because voids cannot be added. assumption: `int` is a subtype of `void`

Type Checking

- Type checking proves facts `e:T`
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each AST node
- In the type rule used for a node `e`:
  - Hypotheses are the proofs of types of `e`'s subexpressions
  - Conclusion is a type of `e`
- Types are computed in a bottom-up pass over the AST

Type Environments

```
-----------------  [false]
  |- false : bool

 s is a character literal
------------------------- [char]
  |- s : char

   |-s:T
---------------------- [address-of]
  |- &s : T *

 s is a string literal
---------------------- [string]
  |- s : char *

    |-b:bool
---------------------- [Not]
  |- !b : bool

    |-x:T
---------------------- [Expr-Stmt]
  |- x=y; : T

 |-x:T  |-y:T
---------------------- [Assignment-Stmt]
  |- x=y : T

 |-e:bool  |-x:T
------------------------- [While-Stmt]
  |- while (e) x : void
```

This is a design decision. One could have said that the type of a while loop is the type of the last statement that executes. But then what if the statement didn't execute even once? Hence a type 'void' means that the statement has a type which is not usable. Similarly for if-then-else: the types in the two branches can be different, so the final type is void.

But what is the type of a variable reference?

```
   x is a variable
  ---------------  [Var]
     |- x:?
```

The local, structural rule does not carry enough information to give `x` a type.

Solution: put more information in the rules!

A *type environment* gives types for *free* variables

- A type environment is a function from ObjectIdentifiers to Types
- A variable is free in an expression if it is not defined within the expression
  - `x` is free in expression `x`
  - `x` is free in expression `x+y`
  - `x` is free in expression `{int y = ...; x+y;}`
  - `y` is a *bound variable* in expression `{int y = ...; x+y;}`

The sentence `O |- e:T` is read

- Under the assumption that free variables have the types given by O, it is provable (|-) that expression `e` has the type `T`

```
i is an integer literal (constant)
---------------------------------  [Int]
    O |- i:int

 O |- e1:int    O |- e2:int
---------------------------------  [Add]
    O |- e1+e2:int
```

Notice that the same assumptions are required in both the hypotheses and the conclusion.

And we can write new rules

```
O(x) = T
---------  [Var]
O |- x:T
```

```
      O[T0/x] |- e1:T1
---------------------------    [Let-No-Init]
 O |- { T0 x; e1 } : T1
```

O[T/x] represents one function, where

```
O[T/x](x) = T
O[T/x](y) = O(y)
```

When we enter the scope, we add a new assumption in our type environment. When we leave the scope, we remove the assumption.

This terminology reminds us of the symbol table. So the type-environment is stored in the symbol table.

- The type environment gives types to the free identifiers in the current scope
- The type environment is passed down the AST from the root towards the leaves
  - To check the type of a new scope, we will see which rules can be applied
  - The Let-Init/Let-No-Init rule will indicate that we should try type-checking the body of the new scope with an updated environment (top-down): O[T/x]
- Types are computed up the AST from the leaves towards the root.

```
      O[T0/x] |- e1:T1
         O |- e2:T0
---------------------------    [Let-Init]
 O |- { T0 x = e2; e1 } : T1
```

Notice that this rule says that the initializer `e2` should have the same type `T0` as `x`. This is quite weak. In general, we should allow such assignment as long as `e2` is of a subtype of `T0`.

Define a relation <= on C++ classes

- X <= X
- X <= Y if X inherits from Y
- X <= Z if X <=Y and Y <= Z

A better version of [Let-Init] using subtyping:

```
      O[T0/x] |- e1:T1
         O |- e2:T0
       T0 <= T
---------------------------    [Let-Init]
 O |- { T x = e2; e1 } : T1
```

A better version of [Assignment-Stmt] using subtyping

```
 O|-x:T0  O|-e1:T1 T1 <=T0
------------------------------ [Assignment-Stmt]
  |- x=e1 : T0
```

(Notice that the type of the assignment statement is T1, so it can participate in more operations than T0)

X?y:z with subtyping. define least-upper-bound. use least-upper-bound to type this ternary operator. Compare with if-then-else

Typing Methods

C++ example

```
        O |- e1:T1
        ...
        O |- en:T1
------------------------------ [Dispatch]
   O |- f(e1,e2,...,en):?
```

Also maintain a mapping for method signatures in O

```
O(f) = (T1,...,Tn,T(n+1))
```

means that there is a method f such that

```
f(x1:T1,x2:T2,x3:T3,...,xn:Tn):T(n+1)
```

Hence, our dispatch rule becomes:

```
        O |- e1:T1
        ...
        O |- en:Tn
        O |- f:(T1',..,Tn',T(n+1)')
        T1<=T1',...Tn<=Tn'
--------------------------------------- [Dispatch]
   O |- f(e1,e2,...,en):T(n+1)
```

Object-oriented languages also implement encapsulation (e.g., some functions are only visible in certain classes, etc.). To handle them, one can extend O to be a function from a tuple of class-name and the identifier-name. Further, to express inheritance, we will use the subtype-relation for the "this" object in the method dispatch.

```
        O |- e0:T0
        O |- e1:T1
```

```
        ...
        O |- en:Tn
        O |- C::f : (T1',..,Tn',T(n+1)')
        T0 < T
        T1<=T1',...Tn<=Tn'
---------------------------------------- [Static-Dispatch]
   O |- e0@T.f(e1,e2,...,en):T(n+1)
```

For object-oriented languages, you also need to know the "current class" C in which the expression is sitting.

The form of a *sentence* in the logic is:

```
O,C |- e:T
```

E.g.,

```
 O,C |- e1:int    O,C |- e2:int
---------------------------------  [Add]
    O,C |- e1+e2:int
```

General themes

- Type rules are defined on the structure of expressions (through structural induction)
- Types of variables are modeled by an environment

Warning: Type rules are very compact! A lot of information in compact rules and it takes time to interpret them.

Implementing Type Checking

- C type checking can be implemented in a single traversal over the AST. This is not true for some other languages like OCaml for example, where multiple passes may be required.
- Type environment is passed down the tree
  - From parent to child
  - Other languages may require multiple passes to construct the type environment at each node
- Types are passed up the tree
  - From child to parent

Let's consider the following rule:

```
 O,C |- e1:int    O,C |- e2:int
---------------------------------  [Add]
    O,C |- e1+e2:int
```

This says that to type-check `e1+e2`, then we have to type-check `e1` and `e2` separately in the same O,C environment.

```
TypeCheck(Environment, e1+e2) = {
  T1 = TypeCheck(Environment, e1);
  Check T1 == int;
  T2 = TypeCheck(Environment, e2);
  Check T2 == int;
  return int;
}
```

Let's consider a more complex example:

```
         O |- e0:T0
      O[T0/x] |- e1:T1
       T0 <= T
----------------------------    [Let-Init]
 O |- { T x = e0; e1 } : T1
```

Let's look at the corresponding type-checking implementation:

```
TypeCheck(Environment, { T x = e0; e1 }) = {
  T0 = TypeCheck(Environment, e0);
  T1 = TypeCheck(Environment.add(x:T), e1);
  Check subtype(T0,T1);
  return T1;
}
```

Static vs. Dynamic Typing

- Static type systems detect common errors at compile time
- But some correct programs are disallowed
  - Some argue for dynamic type checking instead. e.g., Python, Perl, ..
  - Others want more expressive static type checking. e.g., fancier and fancier type systems. But can become quite complex making it harder for programming with such highly-expressive type-systems.

  The dynamic type of an object is the class C that is used in the "`new C`" expression that created it.
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
- The static type of an expression captures all dynamic types the expression could have
  - A compile-time notion

Formalizing the relationship:

Soundness theorem: for all expressions E, dynamic_type(E) = static_type(E)

In all executions, E evaluates to values of the type inferred by the compiler. You have to actually run the program to get the dynamic_type. In the early days of programming languages, this was the type of property that was proved for their type systems.

Consider C++ program:

```
class A { ... }
class B : public A { ... }
void foo() {
  A foo;
  B bar;
  ....
  foo =  bar
}
```

Static type of foo is "A", but the dynamic type of foo is "A" and "B" at different program points.

Soundness theorem for object-oriented languages that allow subtyping.

```
\forall{E}{dynamic_type(E) <= static_type(E)}
```

All operations that can be used on an object of type C can also be used on an object of type C' <= C.

- Such as fetching the value of an attribute
- Or invoking a method on the object

Subclasses *only add* attributes or methods (they never remove attributes/methods).

The dynamic type is obtained through the execution semantics of the program, e.g., if we have an expression `x++`, and the current state of the program determines `x` to be of type `int`, there is a dynamic typing rule that says that the type of the new expression is also `int`. Consider the example `{ T x = e0; x; }` where `e0:T0 <= T`; here the dynamic type of the expression is `T0` (not `T`!). Similarly, the dynamic type of `if x then y:T1 else Y:T2` will be either `T1` or `T2` (not `lub(T1,T2)` or `void`).

In other words, soundness states that the static type system will not accept any incorrect program (that will not pass the dynamic type check of equal power). A dynamic type check for array-bounds is not of equal power (it has more checks enabled); a sound static type check for array-bounds would reject all incorrect programs (but it will also reject a few more that will actually pass the dynamic type check).

Soundness of static type system: all dynamically-type-incorrect programs will be rejected. Ensured by ensuring that static_type is a supertype of dynamic_type. A trivial static-type system that rejects all programs is sound (but not useful).

Completeness: all dynamically-type-correct programs will be accepted. Not possible to ensure in general.

Methods can be redefined but with same type! e.g., C++, Java override.

Example where static type system can be too restrictive

While a static type system may be sound, it may be too restrictive and may disallow certain programs that may be dynamically well-typed.

```
class Count {
  int i = 0; //default value = 0
        Count inc() { i <-- i + 1; return *this; }
};
```

Now consider a subclass `Stock` of `Count`:

```
class Stock : public Count {
  string name; //name of the item
};
```

And the following use of `Stock`:

```
Stock a;
Stock b = a.inc();
... b.name ...
```

This code does not type-check because the return value of `inc` is `Count` which is not a subtype of `Stock`.

This seems like a major limitation as now the derived classes (subtypes) will be unable to use the `inc` method.

`a.inc()` has *dynamic type* `Stock`.

So it is legitimate to write: `Stock b <-- a.inc();`

But this is not well-typed

- `a.inc()` has *static type* `Count`.
- We can extend the type system. Different languages extend the type system in different directions

- Option 1: Use `dynamic_cast`: returns nullptr at runtime if not-successful; else returns a pointer to the object of the new type. Allows bypassing the static type system. May entail runtime cost.
- Option 2: C++ Template : pass type as argument
    - ```
      template<typename T>
      class Count<T> {
        int i = 0;
              T inc() { i <-- i + 1; return *static_cast<T *>(this); } //static_cast gets checked at compile-time!
      }

      class Stock : public Count<Stock>
      {
        ...
      }
      ```
    - Another option: provide `dynamic_cast` operator. Will fail at runtime if the cast cannot be executed successfully. The compiler just treats this as an operator that either returns `nullptr` (cast not successful) or (a pointer to) an element of the new type. In both cases, the compiler can perform the rest of the static type-check using the new type.
- Accept more correct programs
- Increase the expressive power of the type system

Error Recovery

- Detecting where errors occur is easier than in parsing
- Introduce a new type `No_type` for use with ill-typed expressions
- Define `No_type <= C` for all types `C`. Avoids cascading type errors due to one type-error.
- Thus, every operation is defined for `No_type`
    - With a `No_type` result
- The type hierarchy is not a tree anymore, it is a DAG with `No_type` at the bottom
      ```

Runtime Organization

- We have covered the front-end phases: lexing, parsing, semantic analysis
- Now back-end: optimization, code generation
- Before that: we need to talk about runtime organization --- what do we need to generate. Later we will talk about how to generate it (code generation)

Management of runtime resources

- Correspondence between static (compile-time) and dynamic (run-time) structures
  - What is done by the compiler, and what is deferred to the runtime
- Storage organization

Execution of a program is initially under the control of the operating system

- The OS allocates space for a program
- Code is loaded into part of the space
- OS jumps to the entry point (e.g., "main")

Draw a picture of memory with "code" and "other space". Code is usually loaded at one end of the memory space (e.g., low or high end).

- Will draw memory as a rectangle with low address at bottom and high address at top
- Lines delimiting different areas of the memory
- Simplification: assume contiguous memory (need not be necessary)

- Other space = Data space
- Compiler is responsible for
  - Generating code
  - Orchestrating use of data area

Activations

- Two goals
  - Correctness
  - Speed
- Complications in code generation from trying to be fast as well as correct
- Over time, fairly elaborate structures have been developed on how to do code generation and runtime structures

Two assumptions:

- Execution is sequential; control moves from one point in a program to another in a well-defined order
  - Violated in face of concurrency
- When a procedure is called, control always returns to the point immediately after the call
  - Violated in catch/throw style exceptions (an exception may escape multiple procedures before it is caught)
  - Call/cc: call with current continuation
- Even such violating constructs depend on the ideas discussed here

- An invocation of procedure `P` is an *activation* of `P`
- The *lifetime* of an activation of `P` is
  - All the steps to execution `P`
  - Including all the steps in procedures `P` calls
- The *lifetime* of a variable `x` is the portion of execution in which `x` is defined
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

- Observation
  - When P calls Q, then Q returns before P returns
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree
- Consider an example:

```
int g() { return 1; }
int f() { return g(); }

int
main() {
g();
f();
}
```

  Draw the tree of activation lifetimes
  - The example shows nesting and the fact that different calls to the same procedure will have different activations
- Another example

```
int g() { return 1; }
int f(int x) { if (x == 0) then return g(); else return f(x- 1); }

int
main() {
return f(3);
}
```

  Draw the activation tree
- The activation tree depends on run-time behaviour (may be different for different inputs).
- **Since activations are properly nested, a stack can track the currently active activations**
  - The stack will not keep track of the entire activation tree
  - It will only keep track of the currently active activations
  - Work the first example showing the tree and the stack incrementally over runtime
- Let's look at the memory layout, where the first portion is occupied by the code. One of the important structures that goes in the "data area" is the stack of actie activations. The stack typically grows in one direction

Activation Records

What information to keep in an activation? That is what we will call the activation record.

- The information needed to manage one procedure activation is called an *activation record* (AR) or *frame*
- If procedure `F` calls procedure `G`, then `G`'s activation record contains a mix of info about `F` and `G`
- `F` is "suspended" until `G` completes, at which point `F` resumes
- `G's` AR contains information needed to
  - Complete execution of `G`
  - Resume execution of `F`
- Example

```
int g() { return 1; }
int f(int x) { if (x == 0) then return g(); else return f(x- 1); }

int
main() {
return f(3);
}
```

  AR: (1) result , (2) arguments , (3) control-link : pointer to the caller's activation , (4) return address

- Executing this program by hand, show the AR for each function (except main). There can be two return addresses for `f`, and show them as "*" and "**" respectively. These represent addresses of the instructions in memory.
- This stack is not as abstract as a general stack. This stack has an array-like layout, and hence compiler writers will often play tricks to exploit this fact that activation records are adjacent in memory.
- This is only one of many possible AR designs. e.g., many compilers don't use a control link, they rely on the fact that the stack is laid out linearly as an array. Similarly, the return address may be in a register (and not in the stack). The compiler determines the AR. Different compilers may employ different ARs. Some ARs are more efficient than others. Some ARs are easier to generate code for.

- The advantage of placing the result in the first word of the AR is that the caller can find the result at a fixed offset from its own activation (size of its AR + 1).
- Can divide responsibility between caller/callee differently
- Real compilers make careful tradeoffs on which part of the activation frame should be in registers and which part in memory
- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record. Thus AR layout and the code generator must be designed together!

Globals and Heap

- A global variable's lifetime transcends all procedure lifetimes. It has a global lifetime
  - Can't store a global in an activation record
- Global s are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values
- Memory layout: Code, static data, stack (grows downwards)

Heap objects

- A value that outlives the procedure that creates it cannot be kept in the AR
- e.g., new A should survive deallocation of the current procedure's AR.

Usually

- The code area contains object code: fixed size and read only
- Static area contains data (not code) with fixed addresses: fixed size, may be readable or writeable
- Stack contains an AR for each currently active procedure. Each AR usually fixed size, contains locals
- Heap contains all other data: In C, heap is managed through *malloc* and *free*

Both heap and stack grow. Must take care they don't grow into each other. Simple solution: put them at opposite ends of memory and let them grow towards each other. If the two regions start touching each other, then the program is out of memory (or it may request more memory, etc.). This design allows different programs to use these different areas as they see fit, e.g., some programs may have large stacks, other may have large static data regions.

Alignment

Low-level but important detail that every compiler writer needs to be aware of.

- Most modern machines are 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
- Data is *word aligned* if it begins at a word boundary
- Most machines have some alignment restrictions
  - E.g., undefined behaviour if access misaligned data

    - Or dramatic performance penalties for poor alignment (can be up to 10x depending on the machine)
- Example: a string "Hello" takes 5 characters (without a terminating '\0').
    - Show an array of bytes with word-boundaries marked with darker lines
    - To word align next word, add 3 "padding" characters to the string
    - The padding is not part of the string, it's just unused memory

Stack machines

Begin talking about code generation. Stack machines are the simplest model of code generation

In a stack machine

- Only storage is a stack
- An instruction

```
r = F(a1, ..., an):
```

    - Pop the top `n` words off the stack
    - Apply `f`
    - Push the result on the stack
- Example: two instructions `push i` and `add`
- Location of the operands/result is not explictly stated
    - Always the top of the stack
- In contrast to a *register machine*
    - `add` instead of `add r1,r2,r3`
    - More compact programs
    - One reason that Java bytecode uses stack evaluation model
        - In early days, Java was meant for mobility and memory was scarce
- There is an intermediate point between a pure stack machine and a pure register machine
- An *n-register stack machine*: Conceptually, keep the top *n* locations of the stack in the registers
- A one-register stack machine: the one register is called an *accumulator*.
- Advantage of a one-register stack machine:
    - In a pure stack machine
        - An `add` does 3 memory operations (load two arguments and store one result
    - In a one-register stack machine
        - The add does `acc <-- acc + top_of_stack`
        - Just one memory read
- Consider an expression `op(e1,...,en)`
    - Note `e1,..,en` are subexpressions
- For each `ei`, compute `ei` with the result in the accumulator; push the result on the stack
- For `en`, just evaluate into accumulator, do not push on stack.
- Evaluate `op` using values on stack and `en` in accumulator to produce result in accumulator
- e.g., `7+5`: push 7; acc <-- 5; acc <-- acc + [top]
- Invariant: after evaluating an expression `e`, the accumulator holds the value of `e` and the stack is unchanged
    - Important property: expression evaluation preserves the stack
- Another example: 3 + (7 + 5)
    - First evaluate 3
    - Save 3
    - Evaluate 7
    - Save 7
    - Evaluate 5
    - Add
    - Add
- Another important property: the expressions are being evaluated left-to-right. i.e., the evaluation order is left to right which also determines the order of the operands on the stack.

- Some code generation strategies may depend on the evaluation order like this one
- Others may not, e.g., where we had named identifiers (e.g., registers) storing the result for each intermediate expression

Introduction to Code Generation

- We first focus on generating code for a stack machine with accumulator
- We want to run the resulting code on a real machine, e.g., the MIPS/x86 processor (or simulator)
    - The ideas do not change much for abstract machines like LLVM IR abstract machine, except some issues like register-allocation.
- We simulate stack machine instructions using MIPS/x86 instructions and registers.

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC)
- Most operations use registers for operands and results
- Use `load` and `store` instructions to use values in memory
- 32 general purpose registers (32 bits each)
    - We will use only three registers: `$sp`, `$a0`, and `$t1` (a temporary register)

x86 architecture

- Complex Instruction Set Computer (CISC)
- Significantly larger opcode set : 400-odd compared to 40-odd in RISC
- Opcodes often can operate on both registers and/or memory
    - Do not necessarily need separate load/store instructions
- 8 general purpose registers (32 bits each)
    - We will use `%esp`, `%eax`, and `%ecx` (a temporary register)
    - Intel engineers felt that it is better to provide more opcodes and less registers. Use on-chip real-estate for more functional units and logic (by saving space through a shorter register file and its connections).

Need some code-generation invariants

- The accumulator is kept in MIPS register `$a0` (or x86 register `%eax`)
- The stack is kept in memory
    - Grows downwards towards lower addresses
    - Standard convention on both MIPS and x86
- For MIPS
    - The next location of the stack is kept in MIPS register `$sp`. The top of the stack is at address `$sp+4`
- For x86
    - The top of the stack is at address `%esp`. The next location of the stack is at address `%esp-4`.
- The name "`sp`" stands for stack-pointer.

MIPS opcodes (relevant only)

- `lw reg1, offset(reg2)`
    - Load 32-bit word from address `reg2+offset` into `reg1`
- `add reg1, reg2, reg3`
    - `reg1 <-- reg2+reg3`
- `sw reg1, offset(reg2)`
    - Store 32-bit word `reg1` to address `reg2+offset`
- `addiu reg1, reg2, imm`
    - `reg1 <-- reg2+imm`
    - "`u`" means overflow is not checked (overflow means different things for signed and unsigned interpretation of the registers)
- `li reg, imm`
    - `reg <-- imm`

x86 opcodes (relevant only)

- `movl %reg1/(memaddr1)/$imm, %reg2/(memaddr2)`
    - Move 32-bit word from register `reg1` (or address `memaddr1` or the immediate value itself) into `reg2` or to memory address `memaddr2`

    - Captures several opcodes in one mnemonic (load, store, li, move-register, etc.). More powerful than RISC, e.g., MIPS cannot move immediate value directly to memory
- `add %reg1/(memaddr1)/$imm, %reg2/(memaddr2)`
    - `%reg2/(memaddr2) <-- reg1/(memaddr1)/imm + %reg2/(memaddr2)`
    - Overflow is always computed for both signed/unsigned arithmetic. Happens in parallel so not in critical performance path, but switches more transistors (more power)
- `push %reg/(memaddr)/$imm`
    - `(%esp-4) <-- reg/(memaddr)/imm; %esp <-- %esp-4`
- `pop %reg/(memaddr)/$imm`
    - `reg/(memaddr)/imm <-- (%esp); %esp <-- %esp+4`
- push/pop are "higher-level" opcodes: enables faster execution paths for these common operations

The stack-machine code for `7+5` in MIPS:

```
acc <-- 7                       : li $a0, 7
push acc                           : sw $a0, 0($sp)
                                     addiu $sp, $sp, -4
acc <-- 5                       : li $a0, 5
acc < acc + top_of_stack        : lw $t1, 4($sp)
                                   : add $a0, $a0, $t1
pop                                : addiu $sp, $sp, 4
```

The stack-machine code for `7+5` in x86:

```
acc <-- 7                       : movl $7,%eax
push acc                           : pushl %eax
acc <-- 5                       : movl $5, %eax
acc < acc + top_of_stack        : addl (%esp),%eax
pop                                : popl %ecx    #just pop the register to unused register ecx
```

A more optimized version was possible in x86:

```
acc <-- 7                       : push $7
push acc
acc <-- 5                       : mov $5, %eax
acc < acc + top_of_stack        : add (%esp), %eax
pop                                : pop %ecx    #just pop the register to unused register ecx
```

Code Generation 1

A higher-level language

```
P --> D; P | D

D --> def id(ARGS) = E
ARGS --> id, ARGS | id
E --> int | id | if E1 = E2 then E3 else E4 | E1 + E2 | E1 - E2 | id(E1,...,En)
```

The first function definition `f` is the entry point

- The "main" routine

Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
             fib(x - 1) + fib(x - 2)
```

For each expression `e` we generate MIPS code that:

- Computes the value of `e` in `$a0`
- Preserves `$sp` and the contents of the stack

We define a code-generation function `cgen(e)` whose result is the code generated for `e`

- The code to evaluate a constant simply copies it into the accumulator:

  ```
  cgen(i) = li $a0 i
  ```

  - This preserves the stack, as required
  - Color key:
    - RED: compile time
    - BLUE: run time

- ```
  cgen(e1+e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp-4
  cgen(e2)
  lw $t1 r($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  ```

  - Establish the two invariants: the accumulator has the value of this expression, and the value of the stack is unchanged.

  Another (more precise) way to write this:

  ```
  cgen(e1+e2) =
  cgen(e1)
  print "sw $a0 0($sp)"
  print "addiu $sp $sp-4"
  cgen(e2)
  print "lw $t1 r($sp)"
  print "add $a0 $t1 $a0"
  print "addiu $sp $sp 4"
  ```

  This latter syntax is a bit more verbose, so we will largely stick to the short-hand notation.
- "Optimization": Put the result of `e1` directly in `$t1`

  ```
  cgen(e1+e2) =
  cgen(e1)
  move $t1 $a0
  cgen(e2)
  add $a0 $t1 $a0
  ```

  Is this code correct? Why not? Consider if `e2=e3+e4`
- The code for + is a template with "holes" for code for evaluating `e1` and `e2`.
- Stack machine code generation is recursive
  - Code for `e1+e2` is code for `e1` and `e2` glued together
- Code generation can be written as a recursive descent of the AST
  - At least for expressions

- ```
  cgen(e1-e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp-4
  cgen(e2)
  lw $t1 r($sp)
  sub $a0 $t1 $a0
  addiu $sp $sp 4
  ```

Dealing with if-then-else constructs:

- New instruction: `beq reg1 reg2 label`
  - Branch to label if `reg1=reg2`
- New instruction: `b label`
  - Unconditional jump to label

- ```
  cgen(if e1=e2 then e3 else e4) =
  cgen(e1)
  ```
  ```

```
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
addiu $sp $sp 4
beq $a0 $t1 true_branch
...
false_branch:
cgen(e4)
b end_if
true_branch:
cgen(e3)
end_if:
```

Code for function calls and function definitions depends on the layout of the AR

A very simple AR suffices for this language:

- The result is always in the accumulator
  - No need to store the result in the AR
- The activation record holds the actual parameters
  - For `f(x1,...,xn)` push `x1,...,xn` on the stack
  - These are the only variables in this language
- The stack discipline guarantees that on function exit `$sp` is the same as it was on function entry
  - No need for a control link (which is usually needed to find another activation).
- We need the return address
- A pointer to the current activation is useful
  - This pointer lives in the register `$fp` (frame pointer)
- Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices
- Picture: Consider a call to `f(x,y)`, the AR is:

```
FP:
      old fp  |
      y       } AR of f
      x       |
SP-> RA
```

- The *calling sequence* is the instructions (of both caller and callee) to set up a function invocation
- New instruction: `jal label`
  - Jump to label, save address of next instruction in `$ra`
    - Show an example of what the "address of next instruction" means
  - On x86, the return address is stored on the stack by the "call" instruction
- New instruction: `jr reg`
  - Jump to the address in register reg
- Code for the "caller side":

```
  cgen(f(e1,e2,...,en)) =
```

```
sw $fp 0($sp)
addiu $sp $sp -4
cgen(en)
sw $a0 0($sp)
addiu $sp $sp - 4
...
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp - 4
jal f_entry
```

  - The caller saves its value of the frame pointer
  - Then it saves the actual parameters in reverse order
  - Finally the caller saves the return address in register `$ra`
  - The AR so far is `4*n+8` bytes long
- Code for the "callee side":

```
  cgen(def f(x1,x2,...,xn) = e) =
```

```
f_entry:
move $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen(e)
lw $ra 4($sp)
addiu $sp $sp z
lw $fp 0($sp)
jr $ra
```

  - Note: the frame pointer points to the top, not bottom of the frame
  - The callee pops the return address, the actual arguments and the saved value of the frame pointer
    - This is just one example strategy. Another strategy could be that the caller pops these elements off the stack, and restores the frame pointer. Both are valid strategies.
  - *z* = 4*n+8 (size of the activation record)
- Before call:

```
FP -->

SP -->
```

- On entry:

```
FP -->

     old fp
     y
     x
SP--> (this is where the return address will go)
```

- Before exit:

```
FP -->

     old fp
     y
     x
FP--> ra
SP-->
```

- After exit:

```
FP -->

SP -->
```

- Function calls have preserved the invariant that the stack would be exactly the same after the call, as it was at entry to the call

Variable references are the last construct

- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from `$sp`
- One solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let `xi` be the ith (i=1..n) formal parameter of the function for which code is being generated

```
cgen(xi) = lw $a0 z($fp)                    [z = 4*i]
```

  e.g., `x` at `$fp+4` and `y` at `$fp+8`
- Another solution: keep track of the current `$sp` vis-a-vis the value of `$sp` at function entry, and thus appropriately adjust the addresses of all variables. Pro: one less register (no frame pointer needed now)! Con: much harder to read and debug the code now. Also not possible with VLAs (variable length arrays).

Summary

- The activation record must be designed together with the code generator
  - Can be a *convention* (allows compatibility) or can be decided at compile-time on a function-to-function basis, but any decision should account for all possible callers/callees for any callee/caller.
- Code generation can be done by recursive traversal of the AST
- While we discussed code generation for a stack machine, production compilers often do different things
  - Emphasis is on keeping values in registers (optimization)
    - Especially the current stack frame
  - Intermediate results are laid out in the AR (at fixed offsets for direct access). Not pushed and popped from the stack (can be more expensive as it does not allow value re-use)

Example

```
def sumto(x) = if x = 0 then 0 else x + sumto(x-1)
```

- ```
  sumto_entry:
  move $fp $sp
  sw $ra 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 0
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true1

  false1:
  lw $a0 4($fp)
  sw $a0 0($sp)
  addiu $sp $sp -4
  sw $fp 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)              # x
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 1
  lw $t1 4($sp)
  sub $a0 $t1 $a0           # x - 1
  addiu $sp $sp 4
  sw $a0 0($sp)
  addiu $sp $sp -4
  jal sumto_entry
  lw $t1 4($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  b endif1

  true1:
  li $a0 0

  endif1:
  lw $r1 4(esp)
  addiu $sp $sp 12
  lw $fp 0($sp)
  jr $ra
  ```

  - Quite inefficient: pushing and popping on the stack all the time (also called *stack traffic*)
  - Will discuss more efficient code generation techniques and optimizations later
  ```

Temporaries

Let's discuss a better way for compilers to manage temporary values. Idea: keep temporaries in the AR. The code generator must assign a fixed location in the AR for each temporary. (We will later discuss keeping temporaries in registers, which is even more efficient).

```
def fib(x) = if x = 1 then 0 else if x = 2 then 1 else fib(x-1) + fib(x-2)
```

If we first analyze the code to see how many temporaries we need, we can allocate space for that, instead of pushing and popping it from the stack each time.

- One temporary for expression `x=1`
- One temporary for expression `x=2`
- One temporary each for the two recursive calls to `fib`
- One temporary for the sum of the two values returned by the recursive calls
- One temporary for `x-1`
- One temporary for `x-2`

But many of these temporaries are only needed once and not needed after that. So some of the space can be re-used for the subsequent temporaries. In fact, we can evaluate the whole expression using only two temporaries.

Let `NT(e)` be the number of temps needed to evaluate `e`

```
NT(e1+e2)
```

- Needs at least as many temporaries as `NT(e1)`
- Needs at least as many temporaries as `NT(e2) + 1`
  - Need one extra temporary to hold on to value of `e1`
- Space used for temporaries in `e1` can be reused for temporaries in `e2`
- `NT(e1+e2) = max(NT(e1), 1+NT(e2))`

Generalizing, we get a system of equations:

```
NT(e1+e2)       = max(NT(e1), 1+NT(e2))
NT(e1-e2)       = max(NT(e1), 1+NT(e2))
NT(if e1=e2 then e3 else e4) = max(NT(e1),1+NT(e2), NT(e3), NT(e4)) #once the branch is decided we do not need to hang on to e1/e2
NT(id(e1,...,en)) = max(NT(e1),...,NT(en))  #the space for the result for ei is saved in the new activation record that we are building a
NT(int) = 0
NT(id) = 0
```

Looking at our fib example

```
def fib(x) = if (x[0] = 1[0])[1] then 0[0] else if (x[0] = 2[0])[1] then 1[0] else fib((x[0] - 1[0])[1])[1] + fib((x - 2)[1])[1 + 1]
```

Show that the entire expression evaluates to NT=2

Once we know the number of temporaries required, we can add that much space for the AR

- For a function definition f(x1...xn) = e the AR has 2 + n + NT(e) elements
  - The return address
  - The frame pointer
  - n arguments
  - NT(e) locations for intermediate results

AR layout

```
Old FP
xn
..
x1
RA
Temp NT(e)
Temp NT(e) - 1
..
Temp 1
```

Code generation must know how many temporaries are in use at each point

- Add a new argument to code generation
  - the position of the next available temporary
- The temporary area is used like a small, fixed-size stack
  - Instead of pushing and popping at run time, we simply do all that in the compiler (at compile time)
- Previous code

  ```
  cgen(e1+e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  ```

- New code (avoid stack manipulation instructions)

  ```
  cgen(e1+e2, nt) =
  cgen(e1, nt)
  sw $a0 nt($sp)
  cgen(e2, nt+4)
  lw $t1 nt($fp)
  add $a0 $t1 $a0
  ```

Code Generation for Objects

- OO implementation = Basic code generation + more stuff
- OO slogan: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
- This means that code in class A works unmodified for an object of class B

- How are objects represented in memory?
- How is dynamic dispatch implemented?

```
class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int { a <- a + d };
};

class B inherits A {
  b: Int <- 2;
  f(): Int { a };
  g(): Int { a <- a - b };
};

class C inherits A {
  c: Int <- 3;
  h(): Int { a <- a*c };
};
```

For A methods to work correctly in A, B and C objects, attribute `a` must be in the same "place" in each object.

Object layout

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in the object
  - The attribute is in the same place in every object of that class
  - e.g., `this` points to the start offset of the object, and `a` is at offset 10 in all objects of type A, B, C
- When a method is invoked, the object is `this` and the fields are the object's attributes

Example layout

```
Offset   Information stored
0   Class tag
4   Object size
8   Dispatch ptr
12  Attribute 1
16  Attribute 2
...
```

- Class tag is an integer which identifies the class of the object (e.g., A = 1, B = 2, C = 3, ..)
- Object size is an integer: size of object in bytes
- Dispatch ptr is a pointer to a table of methods (more later)
- Attributes in subsequent slots
- Lay out in contiguous memory

Observation: Given a layout for class `A`, a layout for subclass `B` can be defined by *extending* the layout of `A` with additional slots for the additional attributes of `B`.

Leaves the layout of `A` unchanged (`B` is an extension).

Looking at our example:

```
Class A: {0: Atag, 4: 5, 8: *, 12: a, 16: d}
Class B: {0: Btag, 4: 6, 8: *, 12: a, 16: d, 20: b}
Class C: {0: Ctag, 4: 6, 8: *, 12: a, 16: d, 20: c}
```

The offset of an attribute is the same in a class and all of its subclasses

- Any method for an A1 can be used on a subclass A2

Consider layout for An < ... A3 < A2 < A1

```
Header     |A1 object  |            |
A1 attrs   |           |A2 object   |
A2 attrs                |            |A3 object
A3 attrs                             |
...
An attrs
```

OO Code generation

- Consider the dispatch: `e.g()`
  - Straightforward
- Consider the dispatch: `e.f()`
  - If `e` is type A: invoke A's `f`
  - If `e` is type B: invoke B's `f`
  - If `e` is type C: invoke A's `f`

Every class has a fixed set of methods

- including inherited methods

A *dispatch table* indexes these methods

- An array of method entry points
- A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses
  - A compiler can figure out all the methods of a certain class. e.g., for A there are two methods
  - Based on this the compiler can assign a fixed offset for each method. This offset will be identical for all overriding functions of all its subclasses.

Dispatch table layout

```
Class A: {0: fA}
Class B: {0: fB, 4: g}
class C: {0: fA, 4: h}
```

- The dispatch table for class `A` contains only one method
- The tables for `B` and `C` extend the table for `A` to the right
- Because methods can be overriden, the method for `f` is not the same in every class, but is always at the same offset

The dispatch pointer in an object of class `X` points to the dispatch table for class `X`. Every method `f` of class `X` is assigned an offset `Of` in the dispatch table at compile time

To implement a dynamic dispatch `e.f()`, we

- Evaluate `e`, giving an object `x`
- Call D[Of]
  - `D` is the dispatch table for `x`
  - In the call, `this` is bound to `x`

Multiple Inheritance

If `C` is a subclass of two independent classes `A` and `B` simultaneously (`C` inherits from both `A` and `B`), then:

- The object layout involves first laying-out C's header, then laying-out A's header and attributes, then laying-out B's header and attributes, and finally laying-out C's attributes
- For each use of a C object in a context where C is expected:
  - Generate code assuming that `this` points to C's header and the object layout
- For each use of a C object in a context where A is expected:
  - Generate code to convert C's `this` to A's `this` (by adding an offset to reach A's header), and then using the code generation for A.
- For each use of a C object in a context where B is expected:
  - Generate code to convert C's `this` to B's `this` (by adding an offset to reach B's header), and then using the code generation for B.
- For methods:
  - If `C` overrides a method of `A`, we appropriately modify `A`'s dispatch table.
  - Similarly, if `C` overrides a method of `B`, we appropriately modify `B`'s dispatch table.
  - While generating dispatch code in a context where `C` is expected, if it is an overriding method, then index into the appropriate class (e.g., A or B)

Operational Semantics

Semantics Overview

We must specify for every C expression what happens when it is evaluated. (this is not exactly C, but close).

- This is the "meaning" of an expression

The definition of a programming language:

- The tokens --> lexical analysis
- The grammar --> syntactic analysis
- The typing rules --> semantic analysis
- The evaluation rules --> code generation and optimization

We have specified evaluation rules indirectly

- The compilation of a C program to a stack machine
- The evaluation rules of the stack machine

This is a complete description

- Why isn't it good enough?
  - This is an overspecification. We need the exact specification as that would allow more freedom in code generation and optimization. Overspecification restricts these choices

Assembly-language descriptions of language implementations have irrelevant detail

- Whether to use a stack machine or not
- Which way the stack grows
- How integers are represented
- The particular instruction set of the architecture

We need a complete description that is not overly restrictive. We need a high-level way of describing the behaviour of languages.

Another approach: reference implementations. Again, there will be artifacts of the particular way it was implemented that you didn't mean to be a part of the language (over-specification)

Many ways to specify semantics

- All equally precise/powerful in their specification (do not under/over-specify)
- Some more suitable to various tasks than others

Operational semantics

- Describes program evaluation via execution rules
  - on an abstract machine
- Most useful for specifying implementations
- Two types: big-step and small-step
  - Big-step semantics specify the value of the full expression in terms of its constituent expressions. Below, we will use big-step semantics for our language.
  - Small-step semantics specify how a single step (small step) is taken in the evaluation of a program. These are also called reduction semantics. For example, see Norrish's paper on small-step semantics for the C programming language: An abstract dynamic semantics for C

Denotational semantics

- Program's meaning is a mathematical function
  - A mathematical function that maps input to output
  - Elegant approach
  - Adds complexity through functions that we don't really need to consider for the purposes of describing an implementation

Axiomatic semantics

- Program behaviour described via logical formulae
  - If execution begins in state satisfying X, then it ends in state satisfying Y
  - X, Y formulas in logic
- Foundation of many program verification systems. e.g., static analysis systems to verify properties or discover bugs

Operational semantics

Notation: Logical rules of inference, as in type checking

Recall the typing judgement

```
Context |- e:C
```

In the given *context*, expression `e` has type `C`.

We use something similar for evaluation

```
Context |- e:v
```

In the given *context* (the meaning of context is different from the context for typing judgements), expression `e` evaluates to value `v`.

Example

```
Context |- e1: 5
Context |- e2 : 7
-----------------
Context |- e1+e2 : 12
```

The `Context` does not do much in this rule. In general, `Context` may provide mappings from variables to values for the variables used in `e1` and `e2`.

Example

```
y <-- x + 1
```

We track variables and their values with:

- An *environment*: *where* in memory a variable is
- A *store*: *what* is in the memory

For C/C++: environment maps *variables* to *memory locations*
store maps *memory locations* to *values*

A variable environment maps variables to locations

- Keeps track of which variables are in scope
- Tells us where those variables are

```
E = [a:l1, b:l2]
```

Store maps memory locations to values

```
S = [l1 --> 5, l2 --> 7]
```

`S' = S[12/l1]` defines a store S' such that

```
S'(l1) = 12
S'(l) = S(l) if l != l1
```

C++ values are objects

- All objects are instances of some type or class

`X(a1=l1, ..., an=ln)` is a C++ object where

- `X` is the class of the object
- `ai` are the attributes (including inherited ones)
- `li` is the location where the value of `ai` is stored

Notice that the specification is deliberately keeping things as abstract as possible (avoiding over-specification). e.g., the layout of the object is not specified.

There are constant values in C/C++ that do not have associated memory locations; they only have values., e.g., `(int)5`, `(bool)true`, (char const *)"constant string", \ldots

There is a special value `void`:

- No operations can be performed on it
- Concrete implementations might use any representation of `void` value

The evaluation judgement is:

```
s0, E,S |- e:v, S'
```

- Given `s0` the current (this/self) object
- Given `E` the current variable environment
- Given `S` the current store
- If the evaluation of `e` terminates then
  - The value of `e` is `v`
  - And the new store is `S'`

Notice: the current self object `s0` and the current environment `E` does not change by evaluating an expression. These are invariant under evaluation. However, the contents of the memory may change.

Also notice: there is a qualification which says that if the evaluation of `e` terminates. Read as "if `e` terminates..."

"Result" of evaluation is a value and a new store

- New store models the side-effects

Some things don't change

- The variable environment
- The operational semantics allows for non-terminating evaluations

Operational Semantics for C

Start with simple operational-semantic rules and work our way up to more complex rules.

```
------------------------------     [bool-true]
s0,E,S |- true: bool(true), S
```

```
-------------------------------   [bool-false]
s0,E,S |- false: bool(false), S

i is an integer literal
----------------------   [int]
s0,E,S |- i: int(i), S

s is a string literal
---------------------------------  [string-lit]
s0,E,S |- s: (char const *)(s), S

E(id) = lid
S(lid) = v
----------------  [id]
s0,E,S |- id:v,S

-------------------- [this]
s0,E,S|- this: s0, S

s0,E,S |- e:v, S1
E(id) = lid
S2 = S[v/lid]
--------------------------- [assignment]
s0,E,S |- id <-- e:v, S2
```

(e.g., `x <-- 1 + 1`)

```
s0,E,S |- e1:v1, S1
s0,E,S1 |- e2:v2, S2
--------------------------  [add]
s0,E,S |- e1+e2 : v1+v2, S2
```

(notice that the store used while evaluating `e2` includes the side-effects of evaluating `e1`). These stores also dictate the order of evaluation of the expressions --- e1 needs to be evaluated first to get S1 which is needed by evaluation of `e2`

```
s0,E,S |- e1:v1, S1
s0,E,S1 |- e2:v2, S2
---------------------- [stmt]
s0,E,S |- e1;e2 : v2,S2

s0,E,S |- e:v,S1
---------------------- [stmt-block]
s0,E,S |- {e} : v,S1
```

Example: Consider the expression

```
{ X <-- 7 + 5; 4; }
```

Let's say that initially, `s0=s0`, `E=x:l`, `l<-0`. Let's evaluate this expression in this context (start-state/environment/store).

```
   ....
   .....

-----------------------------------------   -------------------------------
s0,[X:l],[l<-0] |- X<-7+5:12,[l<-12}    s0,[x:l],[l:..] |- 4:4,[l:...]
-------------------------------------------------------------------------
s0,[X:l],[l<-0] |- {X <- 7+5; 4 } : 4, [l:12]
```

More evaluation rules

```
s0,E,S |- e1:bool(true),S1
s0,E,S1 |- e2:v,S2
------------------------------------  [ite-true]
s0,E,S |- if e1 then e2 else e3 : v,S2
```

```
s0,E,S |- e1:bool(false),S1
s0,E,S1 |- e3:v,S3
------------------------------------  [ite-false]
s0,E,S |- if e1 then e2 else e3 : v,S3

s0,E,S |- e1 : bool(false), S1
---------------------------------- [while-false]
s0,E,S |- while (e1) {e2} : void, S1

s0,E,S |- e1 : bool(true), S1
s0,E,S1 |- e2 : v, S2
s0,E,S2 |- while e1 {e2} : void, S3
---------------------------------- [while-true]
s0,E,S |- while (e1) {e2} : void, S3
```

Declarations

Partial rule

```
s0,E,S |- e1 : v1, S1
s0,?,? |- e2: v2, S2
-------------------------------------------
s0,E,S |- { Decl(id:T <-- e1); e2 } v2, S2
```

In what context should `e2` be evaluated?

- Environment like `E` but with a new binding of `id` to a fresh location `lnew`.
- Store like `S1` but with `lnew` mapped to `v1`.

We write `lnew = newloc(S)` to say that `lnew` is a location not already used in `S`.

- `newloc` is like the memory allocation function
- Notice that the spec does not say anything about stack, etc. Just that some memory location is allocated that is not already used.

Complete rule

```
s0,E,S |- e1 : v1, S1
lnew = newloc(S1)
s0,E[lnew/id],S1[v1/lnew] |- e2: v2, S2
-------------------------------------------
s0,E,S |- { Decl(id:T <-- e1); e2 } v2, S2
```

Allocation of a new object

Informal semantics of `new T`

- Allocate locations to hold all attributes of an object of class `T`
  - Essentially, allocate a new object
- Set attributes with their default values
- Evaluate the initializers and set the resulting attribute values
- Return the newly allocated object

Let's assume that for each type `A`, there is a default value `D_A`. In reality, C/C++ have a notion of uninitialized variables, and associated undefined behaviour.

- `D_A = void` for any `A` (could have had default values for some types, e.g., `D_int = int(0)`).

For a class `A`, we write

```
class(A) = (a1:T1 <-- e1, a2:T2 <-- e2, ..., an <-- en}
```

where

- `ai` are the attributes (including the inherited ones)
- `Ti` are the attributes' declared types
- `ei` are the initializers
- attributes are listed in the *greatest-ancestor-first* order
    - If C(c1,c2) < B(b1,b2) < A(a1,a2), then

      ```
      class(C) = (a1:.., a2:..., b1:.., b2:.., c1:.., c2:..)
      ```

```
class(T) = (a1:T1 <- e1, ..., an:Tn <- en)
l1 = newloc(S) for i = 1,..,n
v = T(a1=l1,..an=ln)
S1=S[DT1/l1,...,DTn/ln]
E'=[a1:l1, ..., an:ln]
v,E',S1 |- { a1 <-- e1; ... an <-- en; } : vn, S2
-----------------------------------------------------------
s0,E,S |- new T : v,S2
```

Notice that `E'` has nothing to do with `E`, `E'` is due to a completely different scope. Also notice that the evaluation of `e1`, ..., `en` follows the same order as in `class T` (greatest ancestore first). Also notice that the initializers are evaluated with a new value of the self object (`v`). Also notice that `vn` is ignored.

Summarizing:

- The first three steps allocate the object
- The remaining steps initialize it
    - By evaluating a sequence of assignments
- State in which the initializers are evaluated
    - `this` is the current object
    - Only the attributes are in scope (same as in typing)
    - Initial values of attributes are the defaults

Dispatch

Informal semantics of `e0.f(e1,...,en)`

- Evaluate the arguments in order `e1,..., en`
- Let `e0` be the target object
- Let `X` be the dynamic type of the target object
- Fetch from `f` the definition of `f` (with `n` args).
- Create `n` new locations and an environment that maps `f`'s formal arguments to those locations
- Initialize the locations with the actual arguments
- Set `self` to the target object and evaluate `f`'s body

For a class `A` and a method `f` of `A` (possibly inherited):

```
impl(A, f) = (x1, ..., xn, ebody)
```

where

- `xi` are the names of the formal arguments
- `ebody` is the body of the method

Complete rule

```
s0,E,S |- e1 : v1,S1
s0,E,S1 |- e2 : v2,S2
...
s0,E,S(n-1) |- en : vn,Sn
s0,E,Sn |- e0 : v0,S(n+1)
v0 = X(a1=l1, ..., am=lm)
impl(X,f) = (x1,..,xn, ebody)
```

```
lxi = newloc(Sn+1) for i=1..n
E' = [a1:l1, ..., am:lm][x1/lx1,...,xn/lxn]
S(n+2) = S(n+1)[v1/lx1,...,vn/lxn]
v0,E',S(n+2) |- ebody:v,S(n+3)
------------------------------------------- [dispatch]
s0,E,S |- e0.f(e1,...,en) : v, S(n+3)
```

We are interested in the class-tag of `v0` (`X`) and its attributes. Notice that this is the dynamic type of the object v0. (This provides dynamic dispatch).

What are the names in scope of the method `f`: all attributes and all the formal arguments. `E'` assigns locations to all these names.

The construction of `E'` is interesting because it two square brackets: this is because it is possible that a formal parameter may have the same name as an attribute name, and we want to capture this overriding semantics.

The function body is evaluated in an updated store, where the locations of formal arguments are replaced by expression values (call-by-value). Also the self object is v0. Notice that E' has nothing to do with `E` (static scoping). Dynamic scoping may have E' that is dependent on E.

We did not delete the locations `lxi` after finishing the execution of `ebody`. Is that a problem?

Intermediate Language

- A language between the source and the target
- Provides an intermediate level of abstraction
  - More details than the source
  - Fewer details than the target
- Provides an intermediate level of abstraction
  - e.g., the source language has no notion of registers, the IR may have a notion of registers.
  - Why are we using it? Just experience. Some compilers, in fact, choose to have multiple IRs, and thus multiple lowerings.
    - Intuition: make some decisions upfront (source to IR); hopefully these decisions do not make much difference to optimization opportunity, but simplifies reasoning about the final code generation (as the abstraction level is much closer to the target).
    - An IR would usually always loose some information (information that the compiler developer considers extraneous to optimization opportunity). e.g., loop structure converted to gotos. Ideally the abstraction lowering should not loose much information (should not preclude optimization opportunity!).

The design of an IR is an *art* --- hard to say that this is the best possible IR which will allow the best code generation/optimization. We will consider IR which resembles high-level assembly

- Uses register names, but has an unlimited number
- Uses control structures like assembly language
- Uses opcodes but some are higher-level
  - e.g., `push` translates to several assembly instructions
  - Most opcodes correspond directly to assembly opcodes

Each instruction is of the form

```
x = y op z
x = op y
```

- `y` and `z` are registers and constants
- Common form of IR
- This particular IR is also called *three-address code*

The expression `x + y*z` is translated

- t1 = y * z
- t2 = x + t1

In this representation, each subexpression has a "name" : an effect of allowing only one expression at a time.

IR code generation is very similar to assembly code generation. But use any number or IR registers to hold intermediate results.

```
igen(e, t)
```

- code to compute the value of expression `e` in register `t`.

Example:

```
igen(e1+e2, t) =
  igen(e1, t1)  //t1 is a fresh register
  igen(e2, t2)  //t2 is a fresh register
  t = t1 + t2
```

Unlimited number of registers, means IR code generation is simple. Contrast with stack machine, where we were using stack slots to save intermediate results (many instructions to save/restore); here we can just coin a new register name, and save results to it.

LLVM IR is an example of IR. It resembles three-address code, but with usually higher-level opcodes than assembly.

- IR designer's dilemma example: how high-level should the opcodes be? Too low-level may preclude optimization opportunity (assembly opcodes may be higher-level than IR opcodes, e.g., vector instructions). High-level IR opcodes make the IR design large and bulky (starts looking almost like a CISC ISA). Big problem with IR: needs to be designed for all possible ISAs (makes the design decisions even harder). Typical design choice: support as many opcodes as necessary for all the common optimizations on all the common ISAs.
- The same IR may be used for multiple high-level programming languages. e.g., LLVM may be the target for both C and Java programs. Can again increase the complexity of LLVM IR, because we need to try and retain high-level semantics of all supported languages (making them too low-level for simplicity would preclude optimization).
- Yet, if one can design an IR successfully, it is all perhaps worth the effort. Do all the hard work related to optimization once and reap the benefits everywhere (for all languages, and for all ISAs).

Optimization Overview

Most complexity in modern compilers is in the optimizer

- Also by far, the largest phase in terms of compile-time and in terms of source code size (recall: lexing, parsing, semantic analysis, optimization, code generation)

When should we perform optimizations?

- On AST?
  - Pro: Machine independent
  - Con: Too high level
    - Too abstract --- need to have more details to be able to express the "kind" of machine for which the AST needs to be compiled (e.g., register or stack machine or quantum computer)
- On assembly language
  - Pro: exposes optimization opportunities
  - Con: may be too low level, making optimization difficult (need to undo/redo certain decisions)
  - Con: machine dependent
  - Con: must reimplement optimizations when retargetting
- On IR
  - Pro: can be machine independent, if designed well (can represent a large family of machines)
  - Pro: can expose optimization opportunities, if designed well.
  - Con: IR design critical for exposing optimization opportunities.

We will be looking at optimizations on an IR which has the following grammar:

```
P --> S P | S
S --> id := id op id
       | id := op id
       | id := id
       | push id
       | id := pop
       | if id relop id goto L
       | L:
       | jump L
```

- id's are register names
- constants can replace ids
- typical operators: +, -, *

A basic bloack is a maximal sequence of instructions with

- no labels (except at the first instruction), and
- no jumps (except in the last instruction)

Idea:

- Cannot jump into a basic block (except at beginning)
- Cannot jump out of a basic block (except at end)
- A basic block is a single-entry, single-exit, straight-line code segment

Once we reach the start of a basic block, we are guaranteed to execute all instructions in the BB.
Furthermore, the only way into the basic block is through the first statement.

Consider the basic block:

```
1. L:
2.   t := 2*x
3.   w := t + x
4.   if w > 0 goto L'
```

(3) executes only after (2)

- We can change (3) to w := 3 * x
- Can we eliminate (2) as well? Need to be sure that `t` is not used in other basic blocks.

A control-flow graph is a directed graph with

- Basic blocks as nodes
- An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B
  - e.g., the last instruction in A is `jump Lb`
  - e.g., the last instruction in A is `if id relop id then goto Lb`
  - e.g., execution can fall-through from block A to block B

Example control-flow graph:

```
BB1:
  x := 1
  i := 1

BB1-->BB2

BB2:
L:
  x := x * x
  i := i + 1
  if i < 10 goto L

BB2 --> BB2

BB2 --> BB3
```

The body of a method (or procedure) can be represented as a control-flow graph. There is one initial node (entry node). All "return" nodes are terminal.

Optimization seeks to improve a program's resource utilization

- Execution time (most often)
- Code size
- Memory usage
- Network messages sent, disk accesses, etc.
- Power consumption

Optimization should not alter what the program computes

- The answer must still be the same.

For languages like C, there are typically three granularities of optimization

- Local optimizations
  - Apply to a basic block in isolation
- Global optimizations
  - Apply to a control-flow graph (method body) in isolation
- Inter-procedural optimizations
  - Apply across method boundaries

Production compilers do all these types of optimizations. In general, easies to implement local optimizations and hardest to implement inter-procedural optimizations.

Criteria for evaluating an optimization

- Payoff. What is a good payoff?

- Frequency of occurrence
- Ease of Implementation
- Compilation Time

In practice, often a conscious decision is made not to implement the fanciest optimization known. Why?

- Some optimizations are hard to implement
- Some optimizations are costly in compilation time
- Some optimizations have low payoff. But hard to establish the payoff. Often one optimization may trigger another one, and this is hard to predict in advance.
- Many fancy optimizations are all three!

Current state-of-the-art: the goal is "Maximum benefit for minimum cost"

Local Optimization

- The simplest form of optimization
- Optimize one basic block. (No need to analyze the whole procedure body).

Some statements can be deleted

```
x := x + 0
x := x * 1
```

Eliminating these instructions are great optimizations as we are removing an entire instruction.

Some statements can be simplified

```
x := x * 0  =>  x := 0
```

Maybe the instruction on the right is faster than the instruction on the right. More importantly, assigning to a constant allows more cascading optimizations, as we will see later.

Another example:

```
y := y ** 2  => y := y * y
```

Typically, an architecture will not have the exponentiation instruction and so this operator may have to be implemented by an exponentiation loop in software. For the special cases (e.g., exponent = 2), we can simply replace by multiplication.

Another example:

```
x := x * 8  => x := x << 3
x := x * 15  => t := x << 4; x := t - x
```

Replace multiplication by a power-of-two to a shift-left operation. Can also do this for non powers-of-two. On some machines, left-shift is faster than multiply, but not on all! Modern machines have their own "rewriting logic" in hardware that can deal with these special cases really fast. In other words, some of these compiler local optimizations may become less relevant on modern hardware.

All these transformations are examples of *algebraic simplifications*.

Operations on constants can be computed at compile time

- If there is a statement `x := y op z`
- And `y` and `z` are constants.
- Then `y op z` can be computed at compile time.

Examples

- `x:= 2 + 2` can be changed to `x := 4`
- `if 2 < 0 jump L` can be deleted.
- `if 2 > 0 jump L` can be replaced by `jump L`

This class of optimizations is called *constant folding*. One of the most consequential and most common optimizations performed by compilers.

Another important optimization: eliminate unreachable basic blocks (dead code):

- Code that is unreachable from the initial block
  - e.g., basic blocks that are not the target of any jump or "fall through" from a conditional
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects

        - Increased spatial locality

Called dead-code elimination.

Why would unreachable basic blocks occur?

- e.g., `if (DEBUG) then { .... }`
    - Need constant propagation followed with dead-code elimination to remove this whole code.
- Libraries: a library may supply a 100 methods but we are using only a 3 of those methods, the other 97 are dead-code.
- Usually other optimizations may result in more dead code

Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment.

- Rewrite intermediate code in *single assignment* form

  ```
  x := z + y
  a := x
  z := 2 * x
  ```

  Change to

  ```
  b := z + y
  a := b
  x := 2 * b
  ```

More complicated in general, due to loops

If

- Basic block is in single assignment form
- A definition `x :=` is the first use of `x` in a block

Then

- When two assignments have the same rhs, they compute the same value.

Example

```
x := y + z
....
....
w := y + z
```

We can be sure that the values of `x`, `y`, and `z` do not change in the code. Hence, we can replace the assignment to `w` as follows:

```
x := y + z
....
....
w := x
```

This optimization is called *common-subexpression elimination*. This is also another very common and consequential compiler optimization.

If we see `w := x` appears in a block, replace subsequent uses of `w` with uses of `x`

- Assumes single assignment form

Example:

```
b := z + y
a := b
x := 2 * a
```

This can be replaced with

```
b := z + y
a := b
x := 2 * b
```

This is called *copy propagation*. This is useful for enabling other optimizations

- Constant folding. e.g., if the propagated value is a constant
- Dead code elimination. e.g., the copy statement can be eliminated as it is inconsequential.

Example:

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```

gets transformed to

```
a := 5
x := 10
y := 16
t := x >> 4
```

We could have also assigned 160 to `t`. That would be even better.

If `w := rhs` appears in a basic block and `w` does not appear anywhere else in the program, THEN the statement `w := rhs` is dead and can be eliminated.

- Dead: does not contribute to the program's result

In our copy-propagation example, one of the assignments is dead and can be eliminated.

- Each local optimization does little by itself
- Typical optimizations interact
  - Performing one optimization enables another
- Optimizing compilers can be thought of as a *big bag of tricks*.
- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time
  - Certain properties on these transformations (tricks) ensure that we converge (and not oscillate) in this fixed point procedure. e.g., we always *improve* for some notion of improvement. This means that we are uni-directional and can never oscillate.

Example

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e + f
```

Final form

```
a := x * x
f := a + a
g := 6 * f
```

Possible to simplify further, but the compiler may get stuck in "local minima". e.g., should it have changed a + a to 2 * a, it would have had a better optimization opportunity (replacing g = 12*f and eliminating computation of f).

Peephole optimization

- Optimizations can be directly applied to assembly code. Typically for optimizations that got missed at IR stage (perhaps due to local minima problem).
  . At the assembly-level, we have greater visibility into instruction costs and instruction opcodes. E.g., hardware opcodes may be higher-level than IR opcodes.
- Peephole optimization is effective for improving assembly code
  - The "peephole" is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)

Peephole optimizations are usually written as replacement rules

```
i1; i2; i3; ..; in --> j1; j2; ..; jm
```

Example:

```
move $a $b; move $b $a --> move $a $b
```

Works if the second instruction is not the target of a jump (i.e., both instructions belong to a basic block).

Another example:

```
addiu $a $a i; addiu $a $a j --> addiu $a $a (i+j)
```

Many (but not all) of the basic block optimizations can be cast as peephole optimizations

- Example: `addiu $a $b 0` --> `move $a $b`
- Example: `move $a $a` -->
- These two together eliminate `addiu $a $a 0`.

As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect

- Need to ensure that the replacement rules cannot cause oscillations. e.g., each replacement rule can only "improve" the code.

Many simple optimizations can still be applied on assembly language.

"Program optimiztion" is grossly misnamed

- Code "optimizers" have no intention or even pretense of working towards the optimal program.
- Code "improvers" is a better term.

Dataflow Analysis

Recall the simple basic-block optimizations

- Constant propagation
- Dead code elimination

```
X := 3
Y := Z + W
Q := X + Y
```

to

```
X := 3
Y := Z + W
Q := 3 + Y
```

to (if X is not used anywhere else)

```
Y := Z + W
Q := 3 + Y
```

These optimizations can be extended to an entire control-flow graph (CFG).

```
BB1:
X := 3
B > 0 then goto BB2
         else goto BB3

BB2:
Y := Z + W

BB3:
Y := 0

BB4:
A := 2 * X
```

Here we can substitute X with 3 in BB4.

Another example

```
BB1:
X := 3
B > 0 then goto BB2
         else goto BB3

BB2:
Y := Z + W
X := 4

BB3:
Y := 0

BB4:
A := 2 * X
```

Now we cannot replace X with 3 in BB4 even though X is only assigned constants.

How do we know if it is OK to globally propagate constants?

To replace a use of `x` by a constant `k`, we must know: *On every path to the use of `x`, the last assignment to `x` is `x := k`.*

Let's look at the first example. Indeed this condition is satisfied for X:=3 at BB4.

Let's look at the second example. This condition is *not* satisfied for `X` at BB4.

The correctness condition is not trivial to check. "All paths" includes paths around loops and through branches of conditionals.

Checking the condition requires *global dataflow analysis*, i.e., an analysis of the entire CFG.

In general, global optimization tasks share several traits:

- The optimization depends on knowing a property `X` at a particular point in program execution.
- Proving `X` at any point requires knowledge of the entire program.
- It is OK to be conservative. If the optimization requires `X` to be true, then want to know either:
  - `X` is definitely true.
  - Don't know if `X` is true.
  - It is always safe to say "don't know".

Global dataflow analysis is a standard technique for solving problems with these characteristics. Global constant propagation is an example of such a problem.

Global constant propagation

As an example of a general global transformation involving global dataflow analysis.

Recall: To replace a use of x by a constant k, we must know: *On every path to the use of x, the last assignment to x is x := k*. Let's call this condition AA.

Global constant propagation can be performed at any point where AA holds.

Let's first consider the case of computing AA for a single variable X at all program points. One can potentially repeat this procedure for each variable (inefficient but okay; improvements possible by doing all at once -- later).

To make the problem precise, we associate one of the following values with X at every program point:

value	interpretation
\top	This statement never executes (or we have not executed it so far)
C	X = constant C
\bot	X is not a constant

\bot is our safe situation; we can always say that X is not a constant (means that we do not know if it is a constant or not).

```
BB1:
               === X = \bot (at this program point)
X := 3
               === X = 3
B > 0 then goto BB2
         else goto BB3
              === X = 3 on both branches

BB2:
             ==== X = 3
Y := Z + W
             ==== X = 3
X := 4
             ==== X = 4

BB3:
             ==== X = 3
Y := 0
             ==== X = 3

BB4:
             ==== X = \bot
A := 2 * X
```

Given global constant information, it is easy to perform the optimization

- Simply inspect the `x = ?` associated with a statement using x
- If x is a constant at that point, replace all uses of x with that constant.
- But how do we compute `x = ?` at each program point.

The analysis of a compicated program can be expressed as a combination of simple rules relating the change in information between adjacent statements.

The idea is to "push" or "transfer" information from one statement to the next.

For each statement s, we compute information about the value of x immediately before and after s

- Define a function `C` that stands for "constant information".
- `C(x,s,in)` = value of `x` before `s`.
- `C(x,s,out)` = value of `x` after `s`.

Define a *transfer function* that transfers information from one statement to another.

In the following rules, let statement `s` have immediate predecessor statements `p1, ..., pn`.

```
        |
        |
        v
---> s <---
```

1. if C(pi, x, out) = \bot for any i, then C(s, x, in) = \bot. For all we know, the execution may come down that predecessor, and so in that case we can make no prediction about the value of `x`.
2. if C(pi, x, out) = c and C(pi, x, out) = d and c <> d, then C(s, x, in) = \bot. We saw this in the example that we did by hand.
3. If C(pi, x, out) = c or \top for all i, then C(s, x, in) = c. If we come along a path where x=c, this is trivial to see. For a path with \top, it means that this never executes and so it can be ignored.
4. If C(pi, x, out) = \top for all i, then C(s, x, in) = \top. Every predecessor is unreachable, and so x itself is unreachable.

Rules 1-4 relate the *out* of one statement to the *in* of the next statement. Now we need rules relating the *in* of a statement to the *out* of the same statement.

If C(s,x,in)=\top then C(s,x,out)=\top, for all s. Let's call this rule 5.

C(x:=c, x, out) = c if `c` is a constant. This rule (rule 6) has lower priority than the previous rule (rule 5). i.e., this rule applies only if C(x:=c, x, in) = d or \bot.

C(x:=f(...), x, out) = \bot if the value of `x` cannot be determined to be a constant after this statement.

C(y:=..., x, out) = C(y:=..., x, in) if x<>y

Algorithm

- For every entry `s` to the program, set `C(s,x,in) = \bot`
- For every other statement (non-entry), set `C(s,x,in)=C(s,x,out)=\top`.
- Repeat until all points satisfy rules 1-8:
  - Pick `s` not satisfying 1-8 and update using the appropriate rule.

Notice we expect that at some point all rules will be satisfied at all points. This is called the *fixed-point* and the corresponding solution, the *fixed-point solution*.

Let's run the algorithm on this example:

```
BB1:
                === X = \bot (at this program point)
X := 3
                === X = \top
B > 0 then goto BB2
         else goto BB3
              === X = \top

BB2:
              ==== X = \top
Y := Z + W
              ==== X = \top
X := 4
              ==== X = \top

BB3:
              ==== X = \top
```

```
Y := 0
              ==== X = \top

BB4:
              ==== X = \bot
A := 2 * X
```

Some guarantees: the fixed-point solution will satisfy the equations at each program point.

Some un-answered questions: what is the guarantee that this analysis will converge? What is the guarantee that this algorithm will result in the best possible solution for this system of solutions?

Analysis of loops

```
BB1:
X := 3
B > 0 then goto BB2
         else goto BB3

BB2:
Y := Z + W
X := 4

BB3:
Y := 0

BB4:
A := 2 * X
```

Assume there is an edge from BB4 to BB3.

Now, when we are computing C(BB3,x,in), we need to know the value of C(BB4,x,out) and in turn C(BB4, x, in) and so on... And we are in a loop (we get back to C(BB3,x,in)).

- Consider the statement `Y:=0`
- To compute whether `X` is constant at this point, we need to know whether `X` is constant at the two predecessors
  - X:=3
  - A:=2*X
- But info for `A:=2*X` depends on its predecessors including `Y:=0`!

Because of cycles, all points must have values at all times.

Intuitively, assigning some initial value allows the analysis to break cycles.

The initial value \top means "So far as we know, control never reaches this point". The initial value is not what is expected at the end but allows us to get going (by breaking the cycle).

Analyzing the example: if we run the algorithm assuming that C(BB4,x,out) is \top, then we will be able to reach a fixed-point solution.

Orderings

We can simplify the presentation of the analysis by ordering the (abstract) values.

```
\bot < c < \top
```

Drawing a picture with "lower" values drawn lower, we get

```
      \top
  /    / | \  \
... -1  0  1 ...
```

```
  \   \ | /  /
      \bot
```

Notice that this is a partial order; not all elements are comparable to each other. e.g., 0 and 1 are not comparable.

With orderings defined:

- \top is the greatest value, \bot is the least.
  - All constants are in between and incomparable.
- Let *glb* be the greatest-lower bound in this ordering:
  - glb(\top, 1) = 1
  - glb(\bot, \top) = \bot
  - glb(\bot, 1) = \bot
  - glb(1, 2) = \bot
- Rules 1-4 can be written using `glb`:
  - `C(s,x,in) = glb{C(p,x,out) | p is a predecessor of s}`.

Simply saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes (it could oscillate forever for example). The use of glb explains why the algorithm terminates:

- Values start as \top can only decrease.
- \top can change to a constant, and a constant to \bot.
- Thus, `C(s,x,_)` (for every statement, for every variable) can change at most twice.

Also, we maintain the invariant that the value at any intermediate point of the algorithm is always *greater* than the best solution value. This is because we initialize all values to \top. Also, we relax minimally --- assuming this invariant is true for the predecessors, it is guaranteed to be true for the successor.

Thus the constant propagation algorithm is linear in program size. Number of steps per variable = Number of C(...) values computed * 2 = Number of program statements * 4 (two C values per program statement, in and out).

Liveness analysis

Once constants have been globally propagated, we would like to eliminate dead code.

```
BB1:
X := 3
B > 0 then goto BB2
         else goto BB3

BB2:
Y := Z + W

BB3:
Y := 0

BB4:
A := 2 * 3
```

After constant-propagating X at BB4, the assignment to X in BB1 is no longer useful. In other words, X:=3 is dead (assuming X not used elsewhere).

Defining what is used (live) and what is not used (dead).

```
X := 3
X := 4
Y := X
```

- The first value of `x` is *dead* (never used).
- The second value of `x` is *live* (may be used).

- Liveness is an important concept.

A variable x is live at statement s if

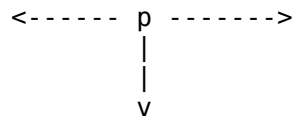
- There exists a statement s' that uses x.
- There is a path from s to s'.
- That path has no intervening assignment to x.

A statement x:=... is dead code if x is dead after the assignment. Dead statements can be eliminated from the program. But we need liveness information first . . .

We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation.

Liveness is simpler than constant propagation, since it is a boolean property (true or false).

Here the set of values is False (definitely not live) and True (may be live). False > True. The glb function is the boolean-OR function in this set of values.



Rule 1: L(p,x,out) = OR{L(s,x,in) | s is a successor of p}. x is live at p if x is live at one of the successor nodes of p.

Rule 2: s: ... := f(x). L(s,x,in) = true if s refers to x on the RHS.

Rule 3: s: x := e where e does not refer to x. L(x:=e,x,in)=false if e does not refer to x. x is dead before an assignment to x because the current value of x at that point will not be used in the future.

Rule 4: L(s,x,in) = L(s,x,out) if s does not refer to x.

Algorithm

1. Let all L(...) = false initially.
2. Repeat until all statements s satisfy rules 1-4
    - Pick s where one of 1-4 does not hold and update using the appropriate rule.

Example:

```
    ==== L(x) = false
x := 0
    ==== L(x) = false
while (x!= 10) {
                ==== L(x) = false -> true (step 1)
  x = x + 1;
    ==== L(x) = false
}
    ==== L(x) = false
return;
    ==== L(x) = false

    ==== L(x) = false
x := 0
                ==== L(x) = false -> true (step 3)
while (x!= 10) {
                ==== L(x) = false -> true (step 1)
  x = x + 1;
    ==== L(x) = false -> true (step 2)
}
    ==== L(x) = false
```

```
return;
    ==== L(x) = false
```

- A value can change from `false` to `true`, but not the other way round. So we use the ordering false > true
- Each value can change only once, so termination is guaranteed.
- Once the analysis is computed, it is simple to eliminate dead code.

Notice that information flowed in the forward direction (in the direction of the program execution) for constant propagation but flowed in the reverse direction (against the direction of the program execution) for liveness analysis. The former types of analyses are called *forward dataflow analyses*. The latter types of analyses are called *backward dataflow analyses*.

Some other analyses that can be modeled as dataflow analyses

- Common-subexpression elimination: maintain a set of *available expressions* of the form x = op(y,z) at each statement. If we see another expression of the type `u = op(y,z)`, replace by `u=x`. Available expressions is a forward data-flow analysis. A value is available at a program location only if it is available at all the predecessor program locations, i.e., the available expressions at a statement is the intersection of the available expressions at the predecessors. In other words, the *meet* operator is intersection. Works best with single-assignment form because can maintain a larger set of available expressions.
- Copy-propagation: an analysis similar to available-expressions, except only for expressions that are of the form x=y. The main difference is in the transformation (substituting variable names vs. substituting expressions by variable names).