Why is it important to know how compilers work?

- To enhance understanding of programming languages
- To write better (more efficient) code in a high-level languages
- To learn techniques that can be useful also in other situations

Different Approaches

- Compiled
- Interpreted
- Hybrid
- JIT-compiled
- Line-By-Line
- ...

All of the above still need lexical analysis and syntactical analysis

Semantics
Static Semantics

- Static semantics is used to describe properties that syntactically valid programs also must have to be semantically valid, e.g., that they are type correct
  - really more related to legal forms of programs rather than meaning
  - some cannot be described by BNF, some just very verbose
  - attribute grammars – add to CFG by carrying some semantic information along inside parse tree nodes

Attribute tree:

\[
\begin{align*}
A_0 &= 9 \\
A_1 &= 8 \\
A_2 &= 3 \\
A_3 &= 5 \\
A_4 &= 1 \\
A_5 &= 5 \\
A_6 &= 1 \\
A_7 &= 9 \\
A_8 &= 8 \\
A_9 &= 3 \\
A_{10} &= 5
\end{align*}
\]

Dynamic Semantics

- Dynamic semantics is used to describe how the meaning of valid programs should be interpreted
  - No single widely acceptable notation or formalism
  - Three common (but not the only) approaches:
    - Operational
    - Denotational
    - Axiomatic

Dynamic Semantics - Operational

- Operational semantics
  - The meaning of a statement defined by describing the effect of running it on a machine
  - Change in the state of the machine defines the meaning of the statement
  - \( (e, \sigma) \rightarrow v \) if the expression \( e \) is evaluated or executed starting in the state \( \sigma \), the resulting computation terminates and yields the result \( v \)

\[
a ::= n \mid X \mid a_0 + a_1 \mid a_0 \cdot a_1 \mid a_0 \ast a_1
\]

\[
\begin{align*}
\langle a_0, \sigma \rangle &\rightarrow n_0 \\
\langle a_1, \sigma \rangle &\rightarrow n_1 \\
\langle a_0 + a_1, \sigma \rangle &\rightarrow n
\end{align*}
\]

where \( n \) is the sum of \( n_0 \) and \( n_1 \)
Dynamic Semantics - Denotational

- Denotational semantics
  - Mathematical denotation of the meaning of the program (typically, a function)
  - The most abstract semantics description method
  - Define a function that maps a program (a syntactic object) to its meaning (a semantic object)
  - Facilitates reasoning about the program, but not always easy to find suitable semantic domains
  
  \[
  [[e]] : \text{States} \rightarrow \text{Values} \quad [[e]](\sigma) = v
  \]
  
  \[
  A[[a]] : \Sigma \rightarrow N
  \]
  
  \[
  [[n]] = \{(\sigma, n) | \sigma \in \Sigma\}
  \]
  
  \[
  [[X]] = \{(\sigma, \sigma(X)) | \sigma \in \Sigma\}
  \]
  
  \[
  [[a_0 + a_1]] = \{(\sigma, n_0 + n_1) | (\sigma, n_0) \in A[[a_0]] \& (\sigma, n_1) \in A[[a_1]]\}
  \]

Denotational vs. Operational

- Denotational semantics is similar to high-level operational semantics, except:
  - Machine is gone
  - Language is mathematics (lambda calculus)

- The difference between denotational and operational semantics:
  - In operational semantics, the state changes are defined by coded algorithms for a virtual machine
  - In denotational semantics, they are defined by rigorous mathematical functions

Ada

Dynamic Semantics - Denotational

- Advantages:
  - Compact & precise, with solid mathematical foundation
  - Can be used to prove the correctness of programs
  - Can be an aid to language design

- Disadvantages
  - Requires mathematical sophistication
  - Hard for programmer to use

- Uses
  - Compiler generation and optimization
Dynamic Semantics - Axiomatic

- Axiomatic semantics
  - Based on formal logic
  - Originally used for formal program verification
  - Define axioms or inference rules for each statement type in the language
  - The inference rules allows transformation of expressions to other expressions
  - The expressions (assertions) state the relationships and constraints among variables that are true at a specific point in execution

- Advantages
  - May be useful in proofs of correctness
  - Solid theoretical foundations
- Disadvantages
  - Predicate transformers are hard to define
  - Hard to give complete meaning
  - Does not suggest implementation
- Uses of Axiomatic Semantics
  - Reasoning about correctness

Dynamic Semantics

- Each form of semantic description has its place:
  - Operational
    - Informal descriptions
    - Compiler work
  - Denotational
    - Formal definitions
    - Provably correct implementations
  - Axiomatic
    - Reasoning about particular properties
    - Proofs of correctness

Back to “Reality”
Parsing

• What is parsing?
  – Check if the input program is correct
  – Produce parse tree or error messages
• Two major approaches
  – Top-down parsing
  – Bottom-up parsing
• Won’t work on all context-free grammars
  – Properties of grammar determine parse-ability
  – We may be able to transform a grammar
Top-Down Parsers -- LL(1), recursive descent

- Start with the root of the parse tree grow toward leaves
  - Root of the tree: node labeled with the start symbol
- Algorithm:
  - Repeat until the fringe of the parse tree matches input string
  - At a node A, select a production for A
    - Add a child node for each symbol on rhs
  - If a terminal symbol is added that doesn’t match, backtrack
  - Find the next node to be expanded (a non-terminal)
- Done when:
  - Leaves of parse tree match input string (success)
  - All productions exhausted in backtracking (failure)

Grammar:

\[
S \rightarrow abc \mid aSQ \\
bQc \rightarrow bbcc \\
cQ \rightarrow Qc
\]

Input string:
aabbcc

21

Algol family

22

Grammar:

\[
S \rightarrow abc \mid aSQ \\
bQc \rightarrow bbcc \\
cQ \rightarrow Qc
\]

Input string:
aabbcc

23

24
Grammar:
\[ S ::= abc | aSQ \]
\[ bQc ::= bbcc \]
\[ cQ ::= Qc \]

Input string:
\[ aabbcc \]

Bottom-Up Parsing
Grammar:
\[ S ::= abc | aSQ \]
\[ bQc ::= bbcc \]
\[ cQ ::= Qc \]

Input string:
\[ aabbcc \]

Assembler
Recursive-Descent Parsing

• There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
• EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

• Assume we have a lexical analyzer named lex, which puts the next token code in nextToken
• The coding process when there is only one RHS:
  – For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  – For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing, cont’d

• A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  – The correct RHS is chosen on the basis of the next token of input (the lookahead)
  – The next token is compared with the first token that can be generated by each RHS until a match is found
  – If no match is found, it is a syntax error

  • Left Recursion Problem
  • Pairwise Disjointness

Lex, Yacc, Antlr
Names, Binding and Scope

- A **name** is a term used for identification
- Most names are identifiers
- Symbols (like ‘+’) can also be names

- A **binding** is an association between two things, such as a name and the thing it names
  - the association of values with identifiers

- The **scope** of a binding is the part of the program (textually) in which the binding is active

Binding Time

- When the “binding” is created or, more generally, the point at which any implementation decision is made
  - language design time, e.g. operator symbols to operations
  - language implementation time, e.g. data type to the range of possible values
  - program writing time, e.g. choose algorithms, data structures and names
  - compile time, e.g. bind a variable to a data type
  - link time, e.g. bind a library call to the subprogram code
  - load time, e.g. bind a static variable to a memory cell
  - run time, e.g. bind a non-static local variable to a memory cell

Static vs Dynamic

- A binding is static if it occurs before run time and remains unchanged throughout program execution
- A binding is dynamic if it occurs during run time and/or can change during execution of the program
Static Type Binding

- Explicit, implicit, inferred
- Advantages
- Disadvantages

Dynamic Type Binding

- Dynamic languages have no types bound to identifiers
- Advantages – there are advantages!
- Disadvantages
  – error detection
  – documentation
  – cost

C and C++

Storage Binding and Lifetime

- Allocation - getting a cell from some pool of available cells
- Deallocation - putting a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell
- Static - bound to memory cells before execution begins and remains bound to the same memory cell throughout execution
- Stack-dynamic - Storage bindings are created for variables when their declaration statements are elaborated
- Explicit heap-dynamic - Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution. Referenced only through pointers or references
- Implicit heap-dynamic - Allocation and deallocation caused by assignment statements
**Scope**

- The scope of a variable is the range of statements over which it is visible.
- The nonlocal variables of a program unit are those that are visible but not declared there.
- The scope rules of a language determine how references to names are associated with variables.

---

**Scope**

- Static scope - with or without nested subprograms
- Blocks - block-structured language
- Declaration order - declarations first (before any code) or anywhere, declarations before use or not
- Global, hiding
- Dynamic Scoping - following execution path
- Advantages Static and Dynamic
- Disadvantages Static and Dynamic
- Scope and Lifetime

---

**Scoping Example**

```
MAIN
 - declaration of x
   - declaration of x
e   call SUB2
   ...
SUB 2
   - reference to x
     ...
call SUB1
   ...
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Static scoping - reference to x is to MAIN's x
Dynamic scoping - reference to x is to SUB1's

---

**The End**
References

• Sebesta, R - Concepts of Programming Languages
• Aho, A., Lam, M., Sethi, R., Ullman, J. - Compilers: Principles, Techniques, and Tools
• Scott, M - Programming Language Pragmatics
• Winskel, G - The Formal Semantics of Programming Languages
• Grune, D & Jacobs, C - Parsing Techniques, a Practical Guide