Dynamic Semantics

CSIT 313

Fall 2015

J.W. Benham

Need for Notation to Describe Dynamic Semantics

- Unlike BNF for syntax, there is no universally accepted notation for dynamic semantics
  - English (or other natural language) is often imprecise
  - Impossible to prove that programs or compilers are correct
- Need for programmers
- Need for compiler writers
- Need for language designers

Operational Semantics

- Based on effects of running program on a machine
- Often based on a simplified machine model
- Example: assignment statement
  - <A> → id = <E>
  - <E> → <E> +< E> | <E >* <E> | (<E>) | id | int-literal

Generating Three-Address Code

- Attributes <A>.code, <E>.code for code generated to make assignment or evaluate expression, respectively
- Attribute <E>.place for identifier to hold value of expression
- Function newTemp() that generates new temporary variables (#T1, #T2, etc.) to hold intermediate results
- Funtion newLabel() that generates new labels (L1, L2, etc.)

Intermediate Code: Three-Address Code

- Each statement performs a single operation such as addition, multiplication, or goto (conditional or unconditional)
  - id = id + id
  - id = id * id
  - id = id
  - id = int-value
  - If(id relop id) goto label
  - goto label

Generating Three-Address Code (2)

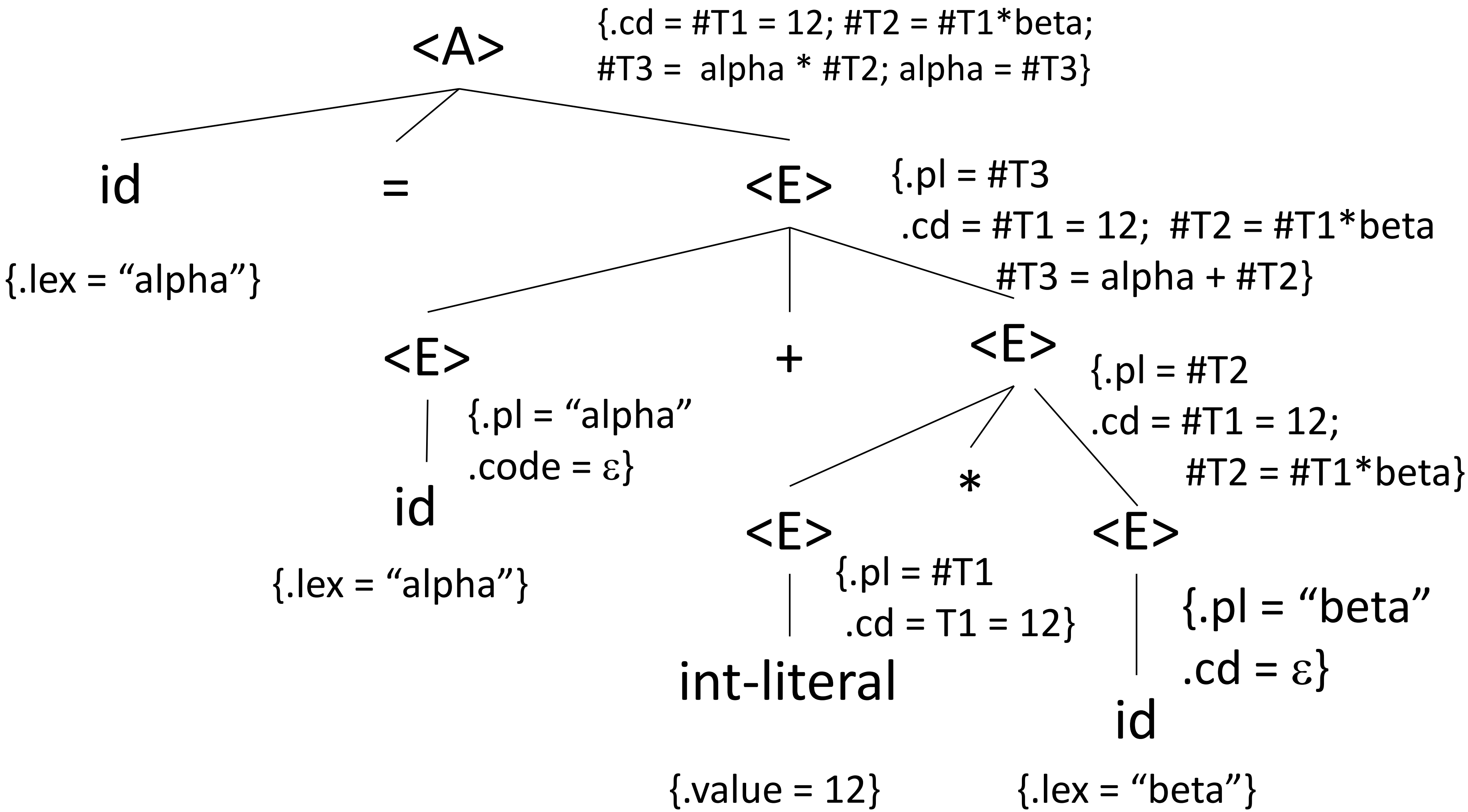
- <A> → id = <E>
  {<A>.code = <E>.code || id.lexeme = E.place}
- <E> → <E>[2] + <E>[3]
  {<E>.place – newTemp()
  <E>.code = <E>[2].code || <E>[3].code ||
  <E>.place = <E>[2].place + <E>[3].place}
- <E> → <E>[2] * <E>[3]
  { // similar to above}

Generating Three-Address Code (3)

- <E> → (<E>[2])
  {<E>.place = <E>[2].place
  <E>.code = <E>[2].code }
- <E> → id
  {<E>.place = id.lexeme <E>.code = ε }
- <E> → int-literal
  { <E>.place = newTemp()
  <E>.code = <E>.place = int-literal.value

Intermediate Code Generation Example

- alpha = alpha + 12 * beta



Logical Pretest Loops

- <S> → <A> {<S>.code = <A>.code}|
  <L> {<S>.code = <L>.code}
- <L> → while (id[1] relop id[2]) <S>
  { <L>.begin = newLabel()
  <L>.end = newLabel()
  <L>.code = <L>.begin:
  if (<id[1].lexeme negation(relop) id[2].lexeme)
  goto L.end ||
  <S>.code || goto <L>.begin || <L>.end: **}**

  – Here, negation(relop) yields the negation of the relational operator (e.g., negation(>=) is <)

- **Classroom exercise:** while(a < b) a = a + 1

Evaluation of Operational Semantics

- Depends on lower-level programming languages
- Can lead to circularities in which concepts are indirectly defined in terms of themselves
- Can be useful for writing compiler front ends
- Corresponds reasonably well with the way programmers think of programs (at least in imperative languages)

Denotational Semantics

- Most rigorous and widely known formal method for describing dynamic semantics
- Solidly based on a mathematical theory: recursive function theory
- Function maps syntactic programming-language constructs to mathematical objects (sets or functions)
  - Domain: **syntactic domain**
  - Range: **semantic domain**

Simple Example: Decimal Numbers

- <dec-num> → ‘0’ | ‘1’ | ’2’ | … | ‘9’ |
  <dec-num> ‘1’ | <dec-num> ‘2’ |
  … <dec-num> ‘9’
- M_{dec}(‘0’) = 0, M_{dec}(‘1’) = 1, … M_{dec}(‘9’) = 9
- M_{dec}(<dec-num> ‘0’) = 10 * M_{dec}(<dec-num>)
- M_{dec}(<dec-num> ‘1’) = 10 * M_{dec}(<dec-num>) + 1
- . . .
- M_{dec}(<dec-num> ‘9’) = 10 * M_{dec}(<dec-num>) + 9

- **Classroom exercise:** Evaluate M_{dec}(“283”)

Program States

- A **program state** is a mapping of the program's variables to values
  - Can be represented as a set of ordered pairs of variable names and the current values of the variables
  - $s = \{<i_1,v_1>, <i_2,v_2>, \ldots , <i_n,v_n>$
  - Special value **undef** for a variable whose value is currently undefined
  - Define a function $\mathrm{VARMAP}(i_j,s) = v_j$
- Functions for program constructs map states to states - reflecting the state change brought about by the construct
  - Some constructs such as expressions are mapped to values

Expressions (1)

We’ll use the following simplified grammar

<expr> → <dec-num> | <var> | <binary-expr>

<binary-expr> → <left-expr> <op> <right-expr>

<left-expr> → <dec-num> | <var>

<right-expr> → <dec-num> | <var>

<op> → + | *

We can now define a function M_e(<expr>,s) for the value of the expression in the given state

Expressions (2)

M_e(<expr>,s) Δ=
case <expr> of
<dec-num> => M_{dec}(<dec-num>)
<var> => **if** VARMAP(<var>,s) == **undef**
then error else VARMAP(<var>,s)
<binary-expr> =>
if (M_e(<left-expr>,s) == **undef** OR
M_e(<right-expr>,s) == **undef**)
then error
else if (<op> == '+')
then M_e(<left-expr>,s) + M_e(<right-expr>,s)
else M_e(<left-expr>,s) * M_e(<right-expr>,s)

Classroom Exercise: For the expression **alpha + 42 , c**ompute M_e(<expr>,s) if s = {<alpha, 53>,…. }

Assignment Statements

M_S(x = <expr>,s) Δ=
if (M_e(<expr>,s) == **error**)
then error
else s' = {<i_1,v_1'>, <i_2,v_2'>, ..., <i_n,v_n'>
where for all j with $1 \le j \le n$
if (i_j == x)
then v_j' = M_e(<expr>,s)
else v_j' = VARMAP($i_{j.}$, s)

Statement Lists

<stmt-list> → <stmt> | <stmt-list>; <stmt>

M_{sl} (<stmt-list>, s) Δ= **case** <stmt-list> of

<stmt> => M_S (<stmt>, s)

<stmt-list> ; <stmt> =>

M_S (<stmt>, M_{sl}(<stmt-list>, s))

Classroom Exercise: Given an initial state

s = {<sum,10>, <prod, 24>, <term,4>} compute

M_{sl} (“**term = term+1; sum = sum + term;**

prod = prod*term”, s)

Logical Pretest Loops

- M_{loop}(**while**(<bool-expr>) {<stmt-list>}, s)
  Δ= **if** M_{bool}(<bool-expr>) == **undef**
  then error
  else if (M_{bool}(<bool-expr> == **false**)
  then s
  else
  M_{loop}(**while**(<bool-expr) {stmt-list} ,
  M_{sl}(<stmt-list>, s))
- **Classroom exercise:** Given the initial state
  s = {<term>,0>, <sum,0>}, compute
  M_{loop}(**while**(term < 3)
  {term = term + 1; sum = sum + term}, s)

Evaluation of Denotational Semantics

- Provides a framework for thinking about programming in a very rigorous way
- Can be used as a tool for evaluating language design
  - If the denotational semantics for a construct are difficult and complex, it may be difficult for language users
- An excellent way to describe a language's semantics concisely
- However, the complexity of denotational semantics makes them of little use to language users

Axiomatic Semantics

- Based on mathematical logic
- Instead of directly defining meaning of program, we focus on what can be proven about the result of program
- Relations among variables and constants that are true for every execution of program
  - Can be used for specifying semantics
  - Probably most useful for **program verification**, i.e., proving that a program is correct

Assertions

- An **assertion** is a **logical predicate** involving the program variables that needs to be true at some point during program execution.
  - Example: {y > 5} x = 2*y -1 {x > 9}
  - An assertion *before* a program statement describes constraints on the values of variables before that statement, and is called a **precondition** of the statement
  - An assertion after a statement describes new constraints on values of variables after the statement has executed and is called a **postcondition** of the statement

Weakest Preconditions

- The **weakest precondition** is the least restrictive precondition that is needed to guarantee the truth of the associated postcondition
- Example: From the preceding slide, $y > 5$ is the weakest precondition that guarantees the postcondition $x > 9$. We could use stronger preconditions (like $y > 6$ or $y > 10$, or $y > 100$).
  – However, if $y \leq 5$, then $2*y - 1 \leq 9$, meaning that $x \leq 9$ after the assignment, so no weaker precondition will work

Assignment Statements

- Given an assignment statement **x = E** and a postcondition **Q**, the weakest precondition is **P = $Q_{x \rightarrow E}$**, the assertion Q with x replaced by E
- **Rules of inference** allows us to infer the truth of one assertion given the truth of other assertions. The are written

$$\frac{S1, S2, S3, \ldots, Sn}{S}$$

  which means that when the **antecedents** S1, S2, S3, …, Sn are all true, we can infer the truth of the **consequent** S.

Assignment Statements and Rules of Inference

- The assertion $\{Q_{x \to E}\}$ x = E $\{Q\}$ is an **axiom**, a logical statement that is assumed to be true
  - It can be written as an inference rule without antecedents: $\dfrac{}{\{Q_{x\to E}\}\ x=E\ \{Q\}}$

- **Rule of consequence:**

$$\frac{\{P\}S\{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

- The rule of consequence allows us to strengthen preconditions or to weaken postconditions

Examples

- **Classroom Exercise:** Determine the weakest preconditions for the following assignment statements and postconditions
  - **x = 4*y – 6 {x > 34}**
  - **x = 3*x + 1 {x ≤ 11}**
- **Classroom Exercise:** Show that the following are true
  - **{y > 8} x = 2*y + 7 {x > 18}**
  - **{x > 4 and y > 3} x = 2*x + y { x > 8}**

Statement Sequences

- Inference rule for a two-statement sequence $\dfrac{\{P1\}S1\{P2\},\{P2\}S2\{P3\}}{\{P1\}S1;S2\{P3\}}$

- Example: Compute the weakest precondition for the following sequence and postcondtion:
  y = 2*x – 7; x = x+y {x > 20}

Conditional Selection

- The inference rule is:

$$\frac{\{B\ and\ P\}S1\{Q\},\{not\ B\ and\ P\}S2\ \{Q\}}{\{P\}\boldsymbol{if}\ B\ \boldsymbol{then}\ S1\ \boldsymbol{else}\ S2\ \{Q\}}$$

- **Classroom Exercise:** Find a precondition (preferably the weakest possible) for
  - **if** (a > b) **then** d = a-b **else** d = b-a {d > 10}
  - **if** (a > b) **then** d = a-b **else** d = b-a {d > 0}

Logical Pretest Loops

- The semantics here require a **loop invariant**: an assertion that is true at the beginning (and end) or every iteration of the loop
- The inference rule is

$$\frac{\{I \ and \ B\} \ S \ \{I\}}{\{I\} \boldsymbol{while} \ (B) \ S \ \{I \ and \ not \ B\}}$$

- Finding a loop invariant is not always easy

Loop Invariant Example

Consider the following code

```
i = 0;
sum = 0;
while (i < n) {
   i = i + 1;
   sum = sum + i*i;
}
// Postcondition: sum = n*(n+1)*(2*n + 1)/6
```