CS 320 : Semantics

Marco Gaboardi

MSC 116

gaboardi@bu.edu

Announcements

- ***Programming Assignment #4*** will be due on Friday, April 3.

- Contrary to the announcement on 17 March 2020: ***TopHat questions*** will be included in Zoom meetings, but not graded.

- All ***Zoom meetings*** will be recorded (by default), and their recordings available for download shortly after the live meetings.

- All ***Zoom links*** for CS 320 are at the bottom of the *Resources* webpage on *Piazza*.

Regular expressions

- A compact way to describe regular grammars:
  - A terminal is a regular expression
  - The or | of two expressions is a regular expression describing two alternatives
  - The grouping (-) of expressions is a regular expression describing sequencing of symbols
  - The quantification * of a regular expression is a regular expression describing zero or more occurrence of the same regular expression

Regular expressions - example

```
<S>  ::=  a<S>
<S>  ::=  b<A>
<A>  ::=  ε
<A>  ::=  c<A>
```


We can describe the grammar above by the following expression.

a*bc*

Regular expressions – another example

How can we describe arbitrary integers through a regular expression?


```
(+|-)(1|2|3|4|5|6|7|8|9|0)*
```

Can we do better?

Regular expressions vs context free grammars

- Regular expressions cannot express everything we can express with context free grammar,
- A regular expression recognizer/generator is much simpler to implement than a parser,
- Regular expressions give potentially infinite vocabularies.

Learning Goals for today

- To understand how semantics information can be integrated in grammars.
- To understand how the dynamic semantics of a program describes the meaning of a program.

Syntax vs Semantics

- Syntax is about "form" and semantics about "meaning".

```
<expr>   ::= <expr> <addop> <term>
           | <term>
<term>   ::= <term> <mulop> <term>
           | <factor>
<factor> ::= <const> | ( <expr> )
<const>  ::= 1|2|3|4|5|6|7|8|9|0
<addop>  ::= + | -
<mulop>  ::= * | /
```


- How do we give meaning to sentences from this grammar?

We have two kinds of semantics: static and dynamic

Semantics: Static vs Dynamic

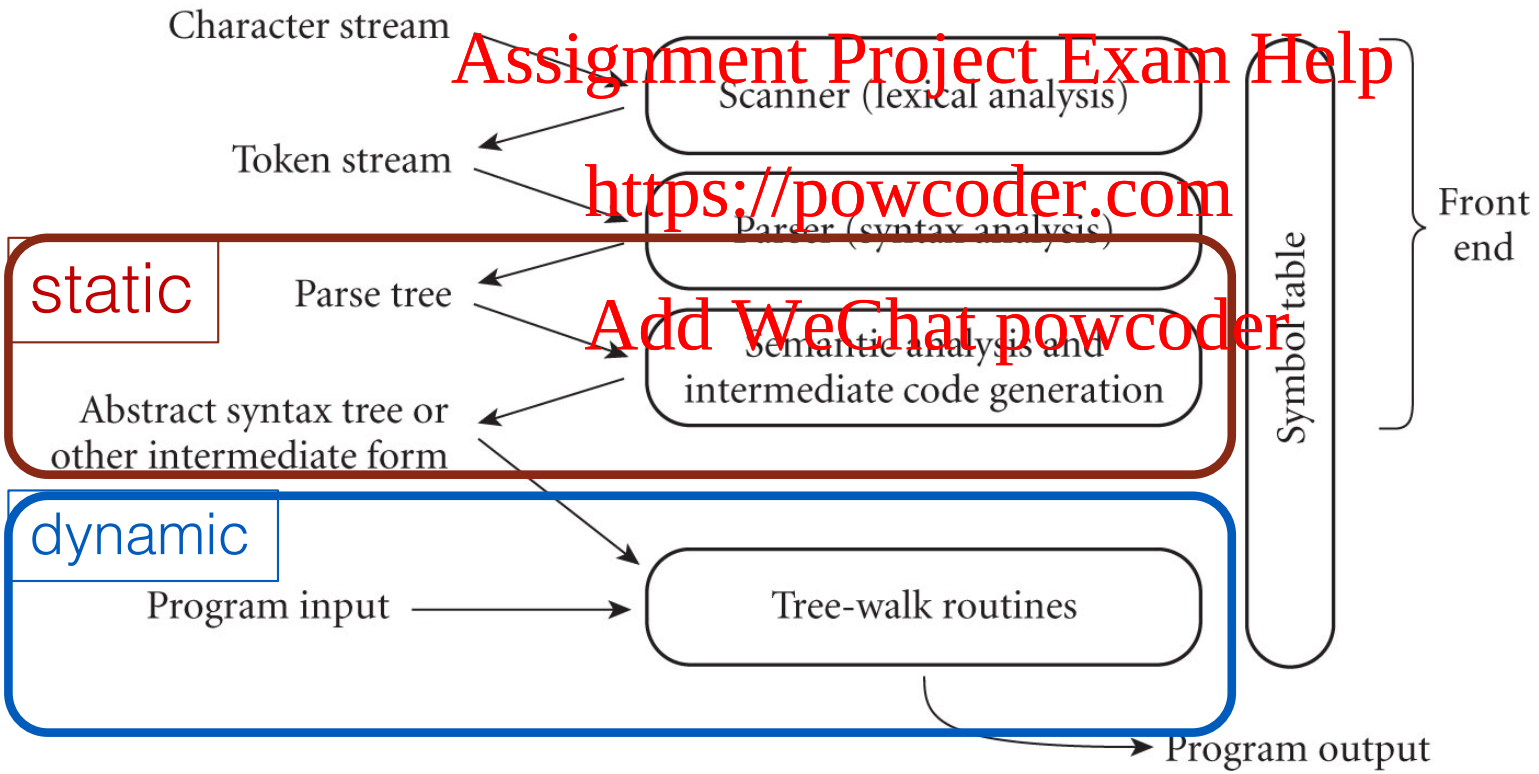
Static semantics:

- Set of rules attaching some high level meaning to the syntactic structure.
  Examples: typing information, well-formedness of commands, etc.
- It is usually enforced statically (before execution).

Dynamic semantics:

- Set of rules describing how the syntactic objects need to be executed.
  Examples: expression evaluation, commands execution, etc.
- It described the way the program must be executed at runtime.

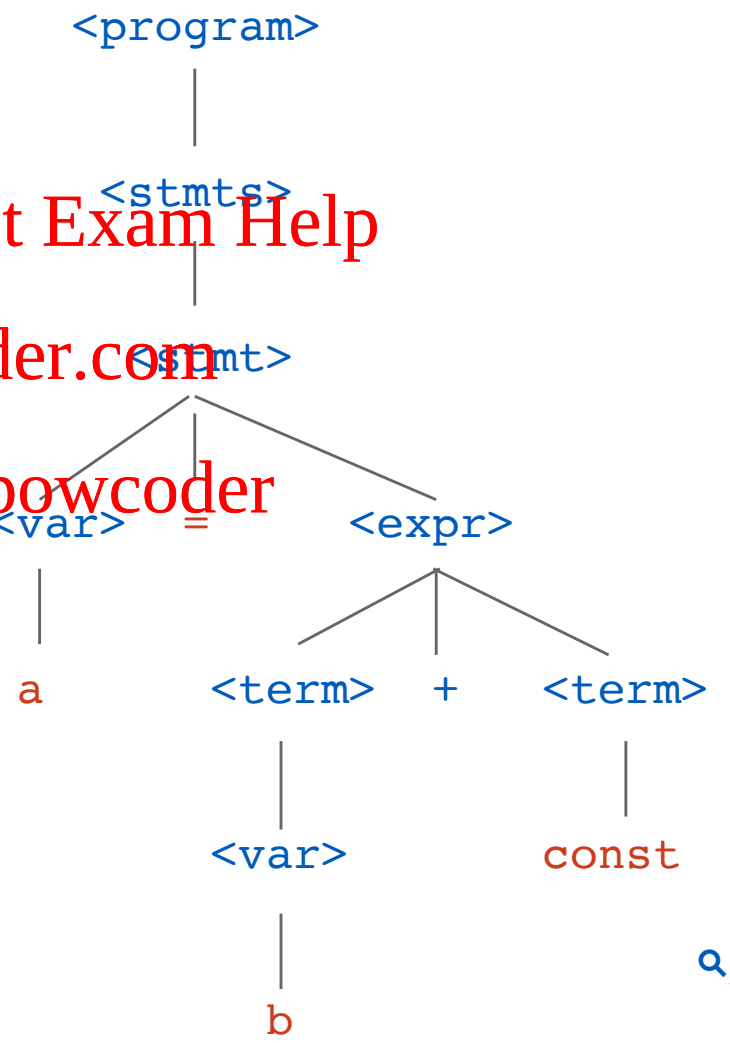
Parsing and semantic analysis



Parse Tree

- A **parse tree** is a hierarchical representation of a derivation

Is this program well-formed?



Examples of well-formedness:

- Is it well typed?
- Are variables declared before being used?
- Do procedure names match?
- ….

Static semantics

- Static semantics enriches the parse tree with additional language features that are difficult or impossible to handle in a BNF/CFG.
- It contributes to turn a parse tree of the input program into a "abstract syntax tree" of its input.
- The abstract syntax tree, abstract away some information of the parse tree, and it respects the rules of the static semantics.

Attribute Grammars

- Introduced by Donald Knuth and Peter Wegner in the 60s-70s.
- CFGs (or BNFs) cannot describe all the important aspects of the syntax of programming languages.

Idea: adding "semantic" attributes to the parse trees to describe these important aspects.

Static Semantics Rules: an example

- Suppose that we have the following BNF rule

```
<proc> ::= procedure <procName> <procBody> end <procName>;
```


- How can we guarantee that he name after "procedure" is the same as the name after "end"?

Problem: BNF cannot capture this for user-defined names.

Solution: associate attributes with symbols and add constraints to the syntactic rule in the grammar.

```
<proc>::=procedure <procName>[1] <procBody> end <procName>[2];

…

<procName>[1].string = <procName>[2].string
```

Static Semantics Rules: another example

- Suppose that we have the following BNF rules

```
<expr> ::= <var> | <expr> + <var>
<var> ::= id
<assgn> ::= <var> := <expr>
```


- How can we enforce the following typing constraints?
  - ids can be either int_type or real_type
  - types of the ids in an expression must be all the same and
  - types in an assignment must match

Static Semantics Rules: another example

```
<expr> ::=  <var>
<expr> ::= <expr> + <var>
<var> ::= id
<assgn> ::= <var> := <expr>
```

How can we enforce the following typing constraints?

- ids can be either int_type or real_type
- types of the ids in an expression must be all the same and
- types in an assignment must match

Possible Solution: associate attributes with symbols and add constraints to the syntactic rule in the grammar.

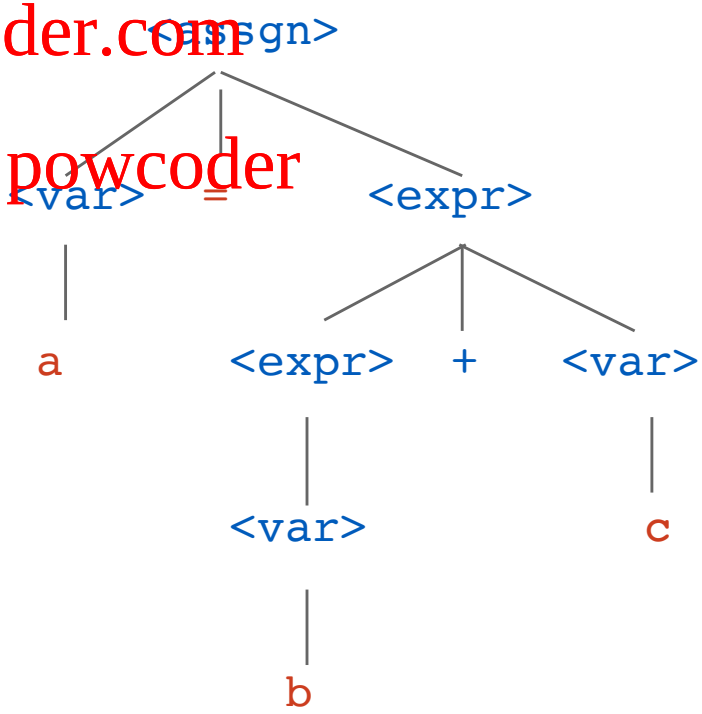
```
<expr>[1] ::=  <var>[1]
<expr>[2] ::= <expr>[3] + <var>[2]
<var>[4] ::= id
<assgn> ::= <var>[5] := <expr>[4]

…

<expr>[2].type = <expr>[3].type + <var>[2].type

<var>[5] = <expr>[4] …
```

Does this work?
What is the problem?

Attributes for internal nodes

- Suppose that we want to enforce typing rules similar to the previous ones.

Problem: the internal nodes are not accessible to the programmer.



Solution: distinguish between attributes that are specified and attributes that are computed.

Attribute Grammars – more formally

Definition: An attribute grammar is a context free grammar were in addition we have:

- For each grammar symbol x there is a set A(x) of attribute values.

- Each rule has a set of functions that define certain attributes of the nonterminals in the rule.
- Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars – example revisited

- Syntactic rule:

```
<expr>[1] ::= <expr>[2] + <var>
```

- Semantic rules:

```
<expr>[1].actual_type ← <var>.actual_type
```

- Predicate:

```
<expr>[2].actual_type = <var>.actual_type
<expr>[1].expected_type = <expr>[1].actual_type
```

BNF vs Attribute grammars

- BNF -- Power to express the structure of a program
  - Properties about the structure
    - Precedences
    - Associativity
- Attribute grammars -- Power to express computation over structure
  - Properties about the semantics
    - Well-formedness
    - Type checking
  - Cannot express arbitrary semantics properties: e.g. guarantee that every variable is initialized with zero.

Semantics: Static vs Dynamic

Static semantics:

- Set of rules attaching some high level meaning to the syntactic structure.
  Examples: typing information, well-formedness of commands, etc.
- It is usually enforced statically (before execution).

Dynamic semantics:

- Set of rules describing how the syntactic objects need to be executed.
  Examples: expression evaluation, commands execution, etc.
- It described the way the program must be executed at runtime.

Dynamic Semantics

Why do we need to specify the semantics?

- Programmers need to know what statements and command mean

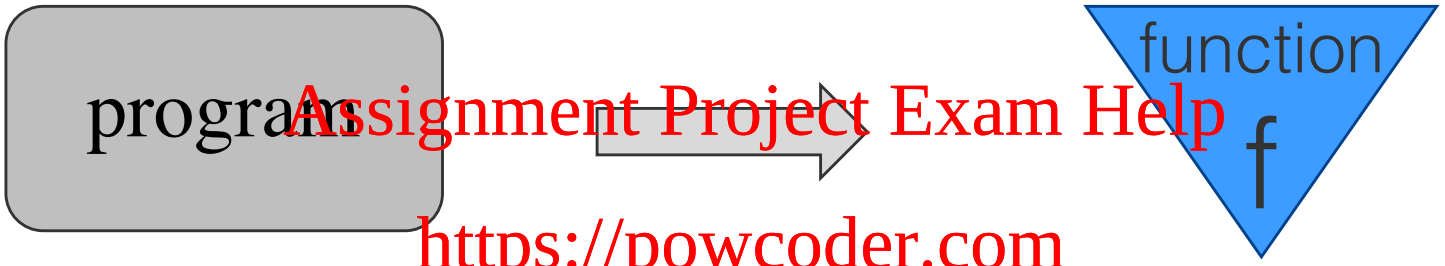
- Compiler writers must know exactly what language constructs do

- Correctness proofs with respect to the specifications,
- Designers need to detect ambiguities and inconsistencies
- …

How would you specify the semantics of a program?

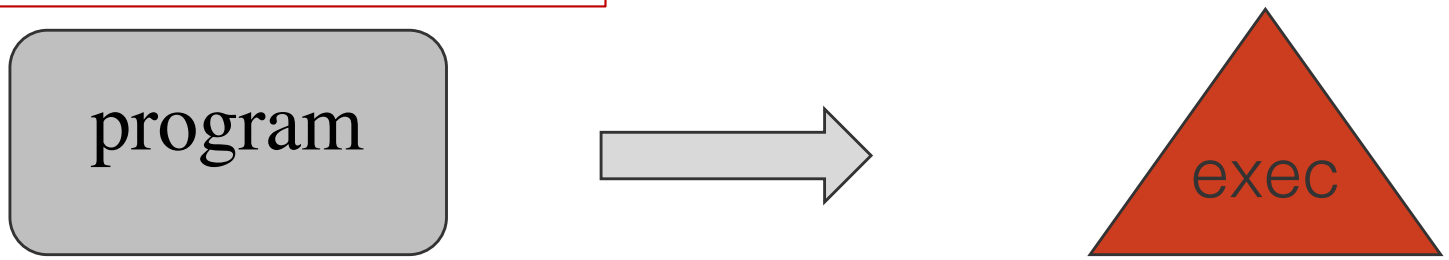
Formal Semantics

Denotational Semantics



A program is described by a mathematical function specifying the input-output relation that the program implements.

Operational Semantics



A program is described by the sequence of transformations that the program implements on the input to produce the output.

Operational Semantics

- Gives the meaning of a program by describing how the statements are executed.
- This can be provided through formal rules or through a description of a machine to execute the programs.
- The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement.

Some examples

- https://docs.oracle.com/javase/specs/jvms/se7/html/
- http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- http://sml-family.org/sml97-defn.pdf
- Our interpreter.

Operational Semantics

- It is usually provided at a level of abstraction that is independent from the machine.
- The detailed characteristics of the particular computer would make actions difficult to describe/understand.
- Different formalism has been developed to describe the operational semantics in a machine-independent way.

We will look into formal rules and derivations.

Language for the Interpreter (simplified)

The language for the interpreter can be described by t
grammar:

```
<const> ::= int | string

<prog>  ::= <com>;quit | <com>;<prog>

<com>   ::= push <const> | add | sub | mul | div
          | neg | rem | swap
```

We use ; here to separate commands, in the interpreter we use a newline.

- A program is a sequence of commands followed by `quit`.
- A command is one the keywords above - in the case of push this is followed by a constant.
- A (simplified) constant is either an int or a string.
- We will denote arbitrary programs with `p,p',…`

Operational semantics for the interpreter

$$(p/S) \rightarrow (p'/S')$$

Here (p/S) is a configuration where p is a program and S is a stack. We call these pairs configurations because we think in terms of an "abstract machine".

We can think about the stack as a list of values (denoted with v):

$$v_n :: \ldots :: v_2 :: v_1 :: []$$

We say that from the configuration (p/S) we can step (or reduce) to the configuration (p'/S') in one step.

An example: `push num`

Informal description: Pushing integers to the stack.

Formal description:

We use p and S because we want to consider all the possible situations.

We use the variable `num` in both sides to represent the same integer.

`(push num;p/S)` → `(p/num::S)`

This says that the program we are processing has the form:
`push num;p`

This says that the stack we are producing has the form:
`num::S`

Another example: `pop`

Informal description: Remove the top value from the stack.

Formal description:

$(\texttt{pop;p}/\texttt{v::S}) \rightarrow (\texttt{p}/\texttt{S})$

We use p and S because we want to consider all the possible situations.

This says that the program we are processing has the form:
`pop;p`

This says that the stack we are producing has the form:
`S`

Does this rule capture all the possible situations?

Another example: `pop` from an empty stack

Informal description: If the stack is empty `:error:` is pushed in the stack.

Formal description:

$$(pop;p/[]) \rightarrow (p/(:error:)::[])$$

We push the error in the stack and we obtain the stack:
`[:error:]`

Another example: `add`

Informal description: it consumes the top two values in the stack, calculate sum and push the result back to the stack.
If one of the following cases occurs, any values popped out from the stack should be pushed back in the same order, then a value :error: should also be pushed onto the stack: 1) only one value in the stack 2) stack is empty 3) not all top two values are integer numbers

Formal description:

What can we do for this case?

$(\texttt{add;p/}v_2\texttt{::}v_1\texttt{::S}) \rightarrow (\texttt{p/}v_2\texttt{+}v_1\texttt{::S})$

$(\texttt{add;p/}v_1\texttt{::[]}) \rightarrow (\texttt{p/(:error:)::}v_1\texttt{::[]})$

$(\texttt{add;p/[]}) \rightarrow (\texttt{p/(:error:)::[]})$

This + represents the addition of the two values

Revisiting the stack

First try: We can think about the stack as a list of values (denoted with `v`):

$v_n :: \ldots :: v_2 :: v_1 :: []$

What is the problem with this?

We don't distinguish values of different types.

Second try: We can think about the stack as a list of typed values (denoted with `type(v)`):

$\texttt{int}(v_n) :: \ldots :: \texttt{string}(v_2) :: \texttt{int}(v_1) :: []$

Revisiting `add`

Informal description: it consumes the top two values in the stack, calculate sum and push the result back to the stack.
If one of the following cases occurs, any values popped out from the stack should be pushed back in the same order, then a value :error: should also be pushed onto the stack: 1) only one value in the stack 2) stack is empty 3) not all top two values are integer numbers

Formal description:

(add;p/int(v_2)::int(v_1)::S) → (p/int(v_2+v_1)::S)

(add;p/type(v_1)::[]) → (p/(:error:)::type(v_1)::[])

(add;p\[]) → (p\(:error:)::[])

(add;p/notint(v)::S)→(p/(:error:)::notint(v)::S)

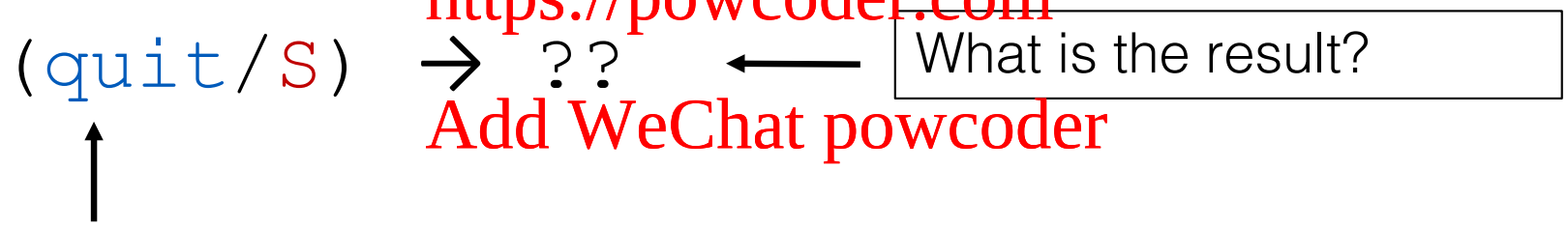
(add;p/int(v_1)::notint(v_2)::S)
→(p/(:error:)::int(v_1)::notint(v_2)::S)

Are we done?

Another example: `quit`

Informal description: he command quit causes the interpreter to stop. Then the whole stack should be printed out to an output file that is specified as the second argument to the interpret function.

Formal description:



We can only have `quit`, with no other command after because of the grammar.

Another example: `quit`

Informal description: he command quit causes the interpreter to stop. Then the whole stack should be printed out to an output file that is specified as the second argument to the interpret function.

Formal description:

(quit/S) → print(S)

This produces the effect of printing S and then stops.

Multiple steps of Operational semantics

We have seen the reduction relation between configurations:

$$(p, S) \rightarrow (p', S')$$

We say that from the configuration (p, S) we can step (or reduce) to the configuration (p', S') in one step.

In general we are interested in (finite or infinite) sequences of reduction steps:

$$(p_1, S_1) \rightarrow (p_2, S_2) \rightarrow (p_3, S_3) \rightarrow \ldots \rightarrow (p_k, S_k)$$

Summary of some rules:

(A) (push num;p/S) → (p/num::S)

(B) (add;p/int(v_2)::int(v_1)::S) → (p/int(v_2+v_1)::S)

(C) (add;p/type(v_1)::[]) → (p/(:error:)::type(v_1)::[])

(D) (add;p\[]) → (p\(:error:)::[])

(E) (quit/S) → print(S)

(F) (add;p/notint(v)::S)→(p/(:error:)::notint(v)::S)

(G) (add;p/int(v_1)::notint(v_2)::S)
→(p/(:error:)::int(v_1)::notint(v_2)::S)

Let's give to each rule a name.

An example:

```
(push 5;push 4;add;quit/S) →
(A)  (push 4;add;quit/int(5)::S) →
(A)  (add;quit/int(4)::int(5)::S) →
(B)  (quit/int(4+5)::S) →
(E)   print(int(4+5)::S)
```

Another example:

```
(push 5;add;quit/[]) →
(A)  (add;quit/int(5)::[]) →
(C)  (quit/(:error:)::int(5)::S) →
(E) print((:error:)::int(5)::S)
```