3: Syntax and Semantics

Modelling Racket computation

A program has a precise meaning and effect.

Modelling Racket computation

A program has a precise meaning and effect.

A model of a programming language provides a way of describing the meaning of a program. Typically this is done informally, by examples.

Modelling Racket computation

A program has a precise meaning and effect.

A model of a programming language provides a way of describing the meaning of a program. Typically this is done informally, by examples.

Students need a model to be able to explain the difference between what actually happened and what they thought should happen.

Modelling Racket computation

A program has a precise meaning and effect.

A model of a programming language provides a way of describing the meaning of a program. Typically this is done informally, by examples.

Students need a model to be able to explain the difference between what actually happened and what they thought should happen.

With Racket, we can do better. Racket has few language constructs, so the model description is short. Mostly what we use to create the model is the language itself.

Modelling Racket computation

A program has a precise meaning and effect.

A model of a programming language provides a way of describing the meaning of a program. Typically this is done informally, by examples.

Students need a model to be able to explain the difference between what actually happened and what they thought should happen.

With Racket, we can do better. Racket has few language constructs, so the model description is short. Mostly what we use to create the model is the language itself.

We won't use diagrams. We won't give vague descriptions of the underlying machine.

Syntax: the way we're allowed to say things.

Syntax and grammar

Syntax: the way we're allowed to say things.

Semantics: the meaning of what we can say.

Syntax: the way we're allowed to say things.

Semantics: the meaning of what we can say.

Example

The English sentence 'Drums fly sadly.'

Syntax: the way we're allowed to say things.

Semantics: the meaning of what we can say.

Example

The English sentence 'Drums fly sadly.'

Syntax in English is specified by grammatical rules. English grammatical rules are not strictly enforced.

Syntax: the way we're allowed to say things.

Semantics: the meaning of what we can say.

Example

The English sentence 'Drums fly sadly.'

Syntax in English is specified by grammatical rules. English grammatical rules are not strictly enforced.

The rules for computer languages are strictly enforced.

An English sentence can be made up of a subject, verb, and object, in that order, or perhaps just a subject and verb.

An English sentence can be made up of a subject, verb, and object, in that order, or perhaps just a subject and verb.

We might express this as follows:

```
sentence = subject verb object
         | subject verb
```

An English sentence can be made up of a subject, verb, and object, in that order, or perhaps just a subject and verb.

We might express this as follows:

```
sentence = subject verb object
         | subject verb
```

The linguist Noam Chomsky formalized grammars in this fashion in the 1950's. The idea proved useful for programming languages.

An English sentence can be made up of a subject, verb, and object, in that order, or perhaps just a subject and verb.

We might express this as follows:

```
sentence = subject verb object
         | subject verb
```

The linguist Noam Chomsky formalized grammars in this fashion in the 1950's. The idea proved useful for programming languages.

The documentation for Beginning Student describes definitions like this:

```
definition = (define (id id id ...) expr)
           | (define id expr)
           | (define-struct id (id ...))
```

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions.

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions. They:

- are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions. They:

- are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.
- must contain at least one non-number.

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions. They:

- are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.
- must contain at least one non-number.
- can't contain spaces or any of these: ( ) , ; { } [ ] ‘ ’ .

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions. They:

- are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.
- must contain at least one non-number.
- can't contain spaces or any of these: ( ) , ; { } [ ] ‘ ’ .

There are spelling rules for numbers (integers, rationals, decimals) which are fairly intuitive.

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions. They:

- are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.
- must contain at least one non-number.
- can't contain spaces or any of these: ( ) , ; { } [ ] ‘ ’ .

There are spelling rules for numbers (integers, rationals, decimals) which are fairly intuitive.

There are some built-in constants, such as `#true` and `#false`.

Spelling rules for Beginning Student

Identifiers (`id`) are the names of constants, parameters, and user-defined functions. They:

- are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.
- must contain at least one non-number.
- can't contain spaces or any of these: ( ) , ; { } [ ] ‘ ’ .

There are spelling rules for numbers (integers, rationals, decimals) which are fairly intuitive.

There are some built-in constants, such as `#true` and `#false`.

Symbols start with a single quote (’) followed by something obeying the rules for identifiers.

More complex rules

Of more interest to us are the rules describing program structure.

A program is a sequence of definitions and expressions.

```
    program = def-or-expr ...

def-or-expr = definition
            | expression
```

More complex rules

Of more interest to us are the rules describing program structure.

A program is a sequence of definitions and expressions.

```
program = def-or-expr ...

def-or-expr = definition
            | expression
```

We have already seen the rules for defining functions and constants.

```
definition = (define (id id id ...) expr)
           | (define id expr)
```

Omission Usage

- An ellipsis (...) is used in English to indicate an **omission**. “The President said that the General was ... wrong.”

Omission Usage

- An ellipsis (...) is used in English to indicate an **omission**. “The President said that the General was ... wrong.”

Pattern Usage

- In mathematics, an ellipsis is often used to indicate a **pattern**. “The positive integers less than n are $1, 2, \ldots, n-1$.”
- In our grammar, an ellipsis is used as a pattern meaning “zero or more repetitions”.

Omission Usage

- An ellipsis (...) is used in English to indicate an **omission**. “The President said that the General was ... wrong.”

Pattern Usage

- In mathematics, an ellipsis is often used to indicate a **pattern**. “The positive integers less than n are $1, 2, \ldots, n-1$.”
- In our grammar, an ellipsis is used as a pattern meaning “zero or more repetitions”.

We will introduce some additional uses of the ellipsis in this course, of both types.

The English sentence “Cans fry cloud” is syntactically correct but has no semantic interpretation.

The English sentence “Cans fry cloud” is syntactically correct but has no semantic interpretation.

The English sentence “Time flies like an arrow” is ambiguous; it has more than one semantic interpretation.

The English sentence “Cans fry cloud” is syntactically correct but has no semantic interpretation.

The English sentence “Time flies like an arrow” is ambiguous; it has more than one semantic interpretation.

We must make sure that our Racket programs are unambiguous (have exactly one interpretation).

A semantic model

A semantic model of a programming language provides a method of predicting the result of running any program.

A semantic model

A semantic model of a programming language provides a method of predicting the result of running any program.

Our model will involve simplification of the program via substitution, as we have been doing all along.

A semantic model

A semantic model of a programming language provides a method of predicting the result of running any program.

Our model will involve simplification of the program via substitution, as we have been doing all along.

Every substitution step yields a valid Racket program, until all that remains is a sequence of definitions and values.

A semantic model

A semantic model of a programming language provides a method of predicting the result of running any program.

Our model will involve simplification of the program via substitution, as we have been doing all along.

Every substitution step yields a valid Racket program, until all that remains is a sequence of definitions and values.

A substitution step finds the leftmost innermost subexpression eligible for rewriting, and rewrites it by the rules we are about to describe.

A semantic model

A semantic model of a programming language provides a method of predicting the result of running any program.

Our model will involve simplification of the program via substitution, as we have been doing all along.

Every substitution step yields a valid Racket program, until all that remains is a sequence of definitions and values.

A substitution step finds the leftmost innermost subexpression eligible for rewriting, and rewrites it by the rules we are about to describe.

$$P_0 \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots \Rightarrow P_n$$

Application of built-in functions in the model

We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions.

```
(+ 3 5) ⇒ 8
(expt 2 10) ⇒ 1024
```

Application of built-in functions in the model

We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions.

```
(+ 3 5) ⇒ 8
(expt 2 10) ⇒ 1024
```

Formally, the substitution rule is:

```
(f v1 ... vn) ⇒ v
```

where `f` is a built-in function and `v` the value of $f(v_1, \ldots, v_n)$.

Application of built-in functions in the model

We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions.

```
(+ 3 5) ⇒ 8
(expt 2 10) ⇒ 1024
```

Formally, the substitution rule is:

```
(f v1 ... vn) ⇒ v
```

where `f` is a built-in function and `v` the value of $f(v_1, \ldots, v_n)$.

Note the two uses of a pattern ellipsis.

Application of user-defined functions in the model

```
(define (term x y)
  (* x (sqr y)))
```

Application of user-defined functions in the model

```
(define (term x y)
  (* x (sqr y)))
```

The function application `(term 2 3)` can be evaluated by taking the body of the function definition and replacing `x` by `2` and `y` by `3`.

Application of user-defined functions in the model

```
(define (term x y)
  (* x (sqr y)))
```

The function application `(term 2 3)` can be evaluated by taking the body of the function definition and replacing `x` by `2` and `y` by `3`.

The result is `(* 2 (sqr 3))`.

Application of user-defined functions in the model

```
(define (term x y)
  (* x (sqr y)))
```

The function application `(term 2 3)` can be evaluated by taking the body of the function definition and replacing `x` by `2` and `y` by `3`.

The result is `(* 2 (sqr 3))`.

The rule does not apply if an argument is not a value, as in the case of the second argument in `(term 2 (+ 1 2))`.

Application of user-defined functions in the model

```
(define (term x y)
  (* x (sqr y)))
```

The function application `(term 2 3)` can be evaluated by taking the body of the function definition and replacing `x` by `2` and `y` by `3`.

The result is `(* 2 (sqr 3))`.

The rule does not apply if an argument is not a value, as in the case of the second argument in `(term 2 (+ 1 2))`.

Any argument which is not a value must first be simplified to a value using the rules for expressions.

Application of user-defined functions in the model

The general substitution rule is:

```
(f v1 ... vn) ⇒ newexp
```

where `(define (f x1 ... xn) exp)` occurs to the left, and `newexp` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `xi` replaced by the value `vi` (for `i` from 1 to n).

Application of user-defined functions in the model

The general substitution rule is:

```
(f v1 ... vn) ⇒ newexp
```

where `(define (f x1 ... xn) exp)` occurs to the left, and `newexp` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `xi` replaced by the value `vi` (for `i` from 1 to n).

Note we are using a pattern ellipsis in the rules for both built-in and user-defined functions to indicate several arguments.

Rewriting using the model

```
(define (term x y) (* x (sqr y)))
(term (- 3 1) (+ 1 2))
```

Rewriting using the model

```
(define (term x y) (* x (sqr y)))
```

```
(term (- 3 1) (+ 1 2))
⇒  (term 2 (+ 1 2))
```

Rewriting using the model

```
(define (term x y) (* x (sqr y)))
```

```
(term (- 3 1) (+ 1 2))
⇒  (term 2 (+ 1 2))
⇒  (term 2 3)
```

Rewriting using the model

```
(define (term x y) (* x (sqr y)))
```

```
(term (- 3 1) (+ 1 2))
⇒  (term 2 (+ 1 2))
⇒  (term 2 3)
⇒  (* 2 (sqr 3))
```

Rewriting using the model

```
(define (term x y) (* x (sqr y)))
```

```
(term (- 3 1) (+ 1 2))
⇒  (term 2 (+ 1 2))
⇒  (term 2 3)
⇒  (* 2 (sqr 3))
⇒  (* 2 9)
```

Rewriting using the model

```
(define (term x y) (* x (sqr y)))
(term (- 3 1) (+ 1 2))
⇒ (term 2 (+ 1 2))
⇒ (term 2 3)
⇒ (* 2 (sqr 3))
⇒ (* 2 9)
⇒ 18
```

Rewriting using the model

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

`id` $\Rightarrow$ `val`, where `(define id val)` occurs to the left.

```
(define x 3) (define y (+ x 1)) y
```

Rewriting using the model

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

`id` $\Rightarrow$ `val`, where `(define id val)` occurs to the left.

```
(define x 3) (define y (+ x 1)) y
⇒ (define x 3) (define y (+ 3 1)) y
```

Rewriting using the model

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

`id` $\Rightarrow$ `val`, where `(define id val)` occurs to the left.

```
(define x 3) (define y (+ x 1)) y
⇒ (define x 3) (define y (+ 3 1)) y
⇒ (define x 3) (define y 4) y
```

Rewriting using the model

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

`id` $\Rightarrow$ `val`, where `(define id val)` occurs to the left.

```
(define x 3) (define y (+ x 1)) y
⇒ (define x 3) (define y (+ 3 1)) y
⇒ (define x 3) (define y 4) y
⇒ (define x 3) (define y 4) 4
```

Substitution in conditional expressions

There are three rules: when the first expression is false, when it is true, and when it is `else`.

```
(cond [false exp] ...) ⇒ (cond ...)
(cond [true exp] ...) ⇒ exp
(cond [else exp]) ⇒ exp
```

Substitution in conditional expressions

There are three rules: when the first expression is false, when it is true, and when it is `else`.

```
(cond [false exp] ...) ⇒ (cond ...)
(cond [true exp] ...) ⇒ exp
(cond [else exp]) ⇒ exp
```

These suffice to simplify any `cond` expression.

Substitution in conditional expressions

There are three rules: when the first expression is false, when it is true, and when it is `else`.

```
(cond [false exp] ...) ⇒ (cond ...)
(cond [true exp] ...) ⇒ exp
(cond [else exp]) ⇒ exp
```

These suffice to simplify any `cond` expression.

Here we are using an omission ellipsis to avoid specifying the remaining clauses in the `cond`.

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

$\Rightarrow$ `(cond [(even? 5) x] [(odd? n) y])`

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

```
⇒  (cond [(even? 5) x] [(odd? n) y])
⇒  (cond [false x] [(odd? n) y])
```

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

```
⇒  (cond [(even? 5) x] [(odd? n) y])
⇒  (cond [false x] [(odd? n) y])
⇒  (cond [(odd? n) y])
```

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

```
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x] [(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
```

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

```
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x] [(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
```

Substitution in conditional expressions

```
(define n 5)
(cond [(even? n) x][(odd? n) y])
```

```
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x] [(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
```

Error: `y` undefined

Errors

A syntax error occurs when a sentence cannot be interpreted using the grammar.

Errors

A syntax error occurs when a sentence cannot be interpreted using the grammar.

A run-time error occurs when an expression cannot be reduced to a value by application of our (still incomplete) evaluation rules.

Errors

A syntax error occurs when a sentence cannot be interpreted using the grammar.

A run-time error occurs when an expression cannot be reduced to a value by application of our (still incomplete) evaluation rules.

Example: `(cond [(> 3 4) x])`

Evaluating this results in a run-time error.

Rewriting Boolean expressions

The simplification rules we use for Boolean expressions involving `and` and `or` are different from the ones the Stepper in DrRacket uses.

Rewriting Boolean expressions

The simplification rules we use for Boolean expressions involving `and` and `or` are different from the ones the Stepper in DrRacket uses.

The end result is the same, but the intermediate steps are different.

Rewriting Boolean expressions

The simplification rules we use for Boolean expressions involving `and` and `or` are different from the ones the Stepper in DrRacket uses.

The end result is the same, but the intermediate steps are different.

The implementers of the Stepper made choices which result in more complicated rules, but whose intermediate steps appear to help students in lab situations.

Rewriting Boolean expressions

The simplification rules we use for Boolean expressions involving `and` and `or` are different from the ones the Stepper in DrRacket uses.

The end result is the same, but the intermediate steps are different.

The implementers of the Stepper made choices which result in more complicated rules, but whose intermediate steps appear to help students in lab situations.

```
(and true exp2 ...) ⇒ (and exp2 ...)
(and false exp2 ...) ⇒ false
(or true exp2 ...) ⇒ true
(or false exp2 ...) ⇒ (or exp2 ...)
(and) ⇒ true
(or) ⇒ false
```

Rewriting Boolean expressions

The simplification rules we use for Boolean expressions involving `and` and `or` are different from the ones the Stepper in DrRacket uses.

The end result is the same, but the intermediate steps are different.

The implementers of the Stepper made choices which result in more complicated rules, but whose intermediate steps appear to help students in lab situations.

```
(and true exp2 ...) ⇒ (and exp2 ...)
(and false exp2 ...) ⇒ false
(or true exp2 ...) ⇒ true
(or false exp2 ...) ⇒ (or exp2 ...)
(and) ⇒ true
(or) ⇒ false
```

As in the rewriting rules for `cond`, we are using an omission ellipsis.

Importance of the substitution model

We can add to the semantic model when we introduce a new feature of Racket.

Importance of the substitution model

We can add to the semantic model when we introduce a new feature of Racket.

Understanding the semantic model is very important in understanding the meaning of a Racket program.

Importance of the substitution model

We can add to the semantic model when we introduce a new feature of Racket.

Understanding the semantic model is very important in understanding the meaning of a Racket program.

Doing a step-by-step reduction according to these rules is called **tracing** a program.

Importance of the substitution model

We can add to the semantic model when we introduce a new feature of Racket.

Understanding the semantic model is very important in understanding the meaning of a Racket program.

Doing a step-by-step reduction according to these rules is called **tracing** a program.

It is an important skill in any programming language or computational system.

Condensed traces

Unfortunately, once we start dealing with unbounded data, traces get very long, and the intermediate steps are hard to make sense of in the Stepper.

Condensed traces

Unfortunately, once we start dealing with unbounded data, traces get very long, and the intermediate steps are hard to make sense of in the Stepper.

In future traces, we will often do a **condensed trace** by skipping several steps in order to put the important transformations onto one slide.

Condensed traces

Unfortunately, once we start dealing with unbounded data, traces get very long, and the intermediate steps are hard to make sense of in the Stepper.

In future traces, we will often do a **condensed trace** by skipping several steps in order to put the important transformations onto one slide.

It is very important to know when these gaps occur and to be able to fill in the missing transformations.

Goals of this module

1. You should intuitively understand the grammar rules used in the Racket documentation to specify the syntax of Racket.

Goals of this module

1. You should intuitively understand the grammar rules used in the Racket documentation to specify the syntax of Racket.
2. You should understand the substitution-based semantic model of Racket, and be prepared for future extensions.

Goals of this module

1. You should intuitively understand the grammar rules used in the Racket documentation to specify the syntax of Racket.
2. You should understand the substitution-based semantic model of Racket, and be prepared for future extensions.
3. You should be able to trace the series of simplifying transformations of a Racket program by hand.